PRACTICAL -13

QUES: *Write a program to perform insertion, deletion and traversal on an AVL Tree.*

CODE:

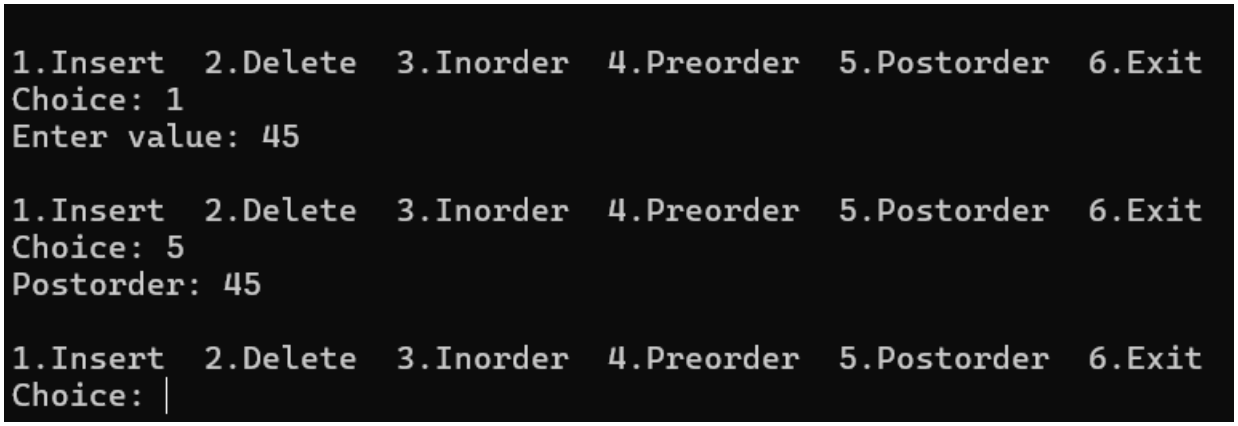
```c
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node *left, *right;
    int height;} Node;
int height(Node *node) {
    return node ? node->height : 0;}
int max(int a, int b) {
    return (a > b) ? a : b;}
Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    node->height = 1;
    return node;}
int getBalance(Node *node) {
    return node ? height(node->left) - height(node->right) : 0;}
Node* rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;}
Node* leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;}
Node* insert(Node* node, int data) {
    if (node == NULL)
        return newNode(data);
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
```

```
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);}
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);}
    return node;}
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;}
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL || root->right == NULL) {
            Node* temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;

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

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);}
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
```

```
        return leftRotate(root);}
    return root;}
void inOrder(Node* root) {
    if (root) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);}}
void preOrder(Node* root) {
    if (root) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);}}
void postOrder(Node* root) {
    if (root) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);}}
void freeTree(Node* root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);}}
int main() {
    Node* root = NULL;
    int choice, data;
    while (1) {
        printf("\n1.Insert  2.Delete  3.Inorder  4.Preorder  5.Postor-
der  6.Exit\n");
        printf("Choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                break;
            case 3:
                printf("Inorder: ");
                inOrder(root);
                printf("\n");
                break;
            case 4:
                printf("Preorder: ");
                preOrder(root);
                printf("\n");
                break;
```

```
            case 5:
                printf("Postorder: ");
                postOrder(root);
                printf("\n");
                break;
            case 6:
                freeTree(root);
                printf("Exiting.\n");
                exit(0);
            default:
                printf("Invalid choice\n");}}
                getch();
    return 0;}
```

OUTPUT:

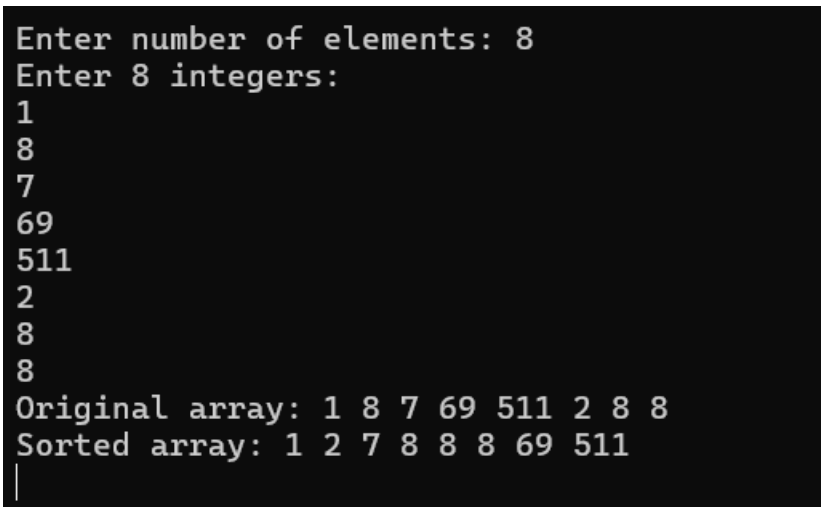


PRACTICAL-14

QUES: *Write a program to implement shell sort..*
CODE:

```
#include <stdio.h>
void shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];}
            arr[j] = temp;}}}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);}
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);}
    printf("Original array: ");
    printArray(arr, n);
    shellSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    getch();
    return 0;}
```

OUTPUT:



PRACTICAL-15

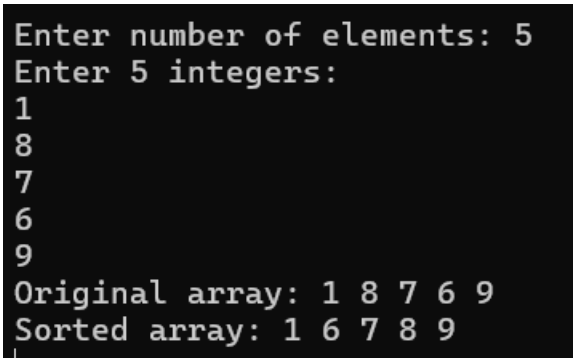
QUES: *Write a program to implement Merge Sort.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for(i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for(j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;}
        k++;}
    while(i < n1) {
        arr[k] = L[i];
        i++;
        k++;}
    while(j < n2) {
        arr[k] = R[j];
        j++;
        k++;}}
void mergeSort(int arr[], int l, int r) {
    if(l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);}}
void printArray(int arr[], int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
```

```
    printf("Original array: ");
    printArray(arr, n);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;}
```

OUTPUT:



PRACTICAL-16

QUES: *Write a program to implement Quick Sort.*
CODE:

```c
#include <stdio.h>
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for(int j = low; j <= high - 1; j++) {
        if(arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);}}
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);}
void quickSort(int arr[], int low, int high) {
    if(low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);}}
void printArray(int arr[], int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;}
```

OUTPUT:

```
Enter number of elements: 4
Enter 4 integers:
8
6
8
7
Original array: 8 6 8 7
Sorted array: 6 7 8 8
```

PRACTICAL-17

QUES: *Write a program to implement BFS and DFS.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
typedef struct {
    int items[MAX_VERTICES];
    int front;
    int rear;} Queue;
typedef struct {
    int items[MAX_VERTICES];
    int top;} Stack;
typedef struct {
    int vertices;
    bool adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];} Graph;
Queue* createQueue() {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = -1;
    q->rear = -1;
    return q;}
bool isEmpty(Queue* q) {
    return q->rear == -1;}
void enqueue(Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1)
        printf("Queue is full\n");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;}}
int dequeue(Queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            q->front = -1;
            q->rear = -1;}
        return item;}}
Stack* createStack() {
    Stack* s = (Stack*)malloc(sizeof(Stack));
    s->top = -1;
    return s;}
bool isStackEmpty(Stack* s) {
    return s->top == -1;}
```

```c
void push(Stack* s, int value) {
    if (s->top == MAX_VERTICES - 1)
        printf("Stack is full\n");
    else {
        s->top++;
        s->items[s->top] = value;}}
int pop(Stack* s) {
    if (isStackEmpty(s)) {
        printf("Stack is empty\n");
        return -1;
    } else {
        int item = s->items[s->top];
        s->top--;
        return item;}}
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->vertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            graph->adjacencyMatrix[i][j] = false;}}
    return graph;}
void addEdge(Graph* graph, int src, int dest) {
    graph->adjacencyMatrix[src][dest] = true;
    graph->adjacencyMatrix[dest][src] = true;}
void BFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    Queue* q = createQueue();
    visited[startVertex] = true;
    printf("BFS traversal starting from vertex %d: ", startVertex);
    printf("%d ", startVertex);
    enqueue(q, startVertex);
    while (!isEmpty(q)) {
        int currentVertex = dequeue(q);
        for (int i = 0; i < graph->vertices; i++) {
            if (graph->adjacencyMatrix[currentVertex][i] && !visited[i]) {
                printf("%d ", i);
                visited[i] = true;
                enqueue(q, i);}}}
    printf("\n");
    free(q);}
void DFSRecursive(Graph* graph, int vertex, bool visited[]) {
    visited[vertex] = true;
    printf("%d ", vertex);
    for (int i = 0; i < graph->vertices; i++) {
        if (graph->adjacencyMatrix[vertex][i] && !visited[i])
            DFSRecursive(graph, i, visited);}}
void DFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    printf("DFS traversal starting from vertex %d: ", startVertex);
    DFSRecursive(graph, startVertex, visited);
    printf("\n");}
```

```
void DFSIterative(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    Stack* s = createStack();
    printf("Iterative DFS traversal starting from vertex %d: ", startVertex);
    push(s, startVertex);
    while (!isStackEmpty(s)) {
        int currentVertex = pop(s);
        if (!visited[currentVertex]) {
            printf("%d ", currentVertex);
            visited[currentVertex] = true;}
        for (int i = graph->vertices - 1; i >= 0; i--) {
            if (graph->adjacencyMatrix[currentVertex][i] && !visited[i]) {
                push(s, i);}}}
    printf("\n");
    free(s);}
int main() {
    Graph* graph = createGraph(7);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 5);
    addEdge(graph, 2, 6);
    printf("Graph Traversal Algorithms\n");
    printf("==========================\n\n");
    BFS(graph, 0);
    DFS(graph, 0);
    DFSIterative(graph, 0);
    free(graph);
    getch();
    return 0;}
```

OUTPUT:

```
Graph Traversal Algorithms
==========================

BFS traversal starting from vertex 0: 0 1 2 3 4 5 6
DFS traversal starting from vertex 0: 0 1 3 4 2 5 6
Iterative DFS traversal starting from vertex 0: 0 1 3 4 2 5 6
```

PRACTICAL-18

QUES: *Write a program to perform Hashing using different resolution techniques.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TABLE_SIZE 10
#define DELETED_NODE (struct DataItem*)(0xFFFFFFFFFFFFFFFL)
struct DataItem {
    int key;
    int data;};
struct LinkedListNode {
    int key;
    int data;
    struct LinkedListNode* next;};
struct DataItem* hashArrayLP[TABLE_SIZE];
struct DataItem* hashArrayQP[TABLE_SIZE];
struct DataItem* hashArrayDH[TABLE_SIZE];
struct LinkedListNode* hashArraySC[TABLE_SIZE];
int hashCode1(int key) {
    return key % TABLE_SIZE;}
int hashCode2(int key) {
    return 7 - (key % 7);}
void initializeHashTables() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashArrayLP[i] = NULL;
        hashArrayQP[i] = NULL;
        hashArrayDH[i] = NULL;
        hashArraySC[i] = NULL;}}
void insertLP(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    while (hashArrayLP[hashIndex] != NULL && hashArrayLP[hashIndex] != DE-
LETED_NODE) {
        if (hashArrayLP[hashIndex]->key == key) {
            hashArrayLP[hashIndex]->data = data;
            free(item);
            return;}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;}
    hashArrayLP[hashIndex] = item;}
void insertQP(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    int i = 0;
    while (hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] != NULL &&
           hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] != DELETED_NODE) {
```

```
        if (hashArrayQP[(hashIndex + i*i) % TABLE_SIZE]->key == key) {
            hashArrayQP[(hashIndex + i*i) % TABLE_SIZE]->data = data;
            free(item);
            return;}
        i++;}
    hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] = item;}
void insertDH(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    while (hashArrayDH[hashIndex] != NULL && hashArrayDH[hashIndex] != DE-
LETED_NODE) {
        if (hashArrayDH[hashIndex]->key == key) {
            hashArrayDH[hashIndex]->data = data;
            free(item);
            return;}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;}
    hashArrayDH[hashIndex] = item;}
void insertSC(int key, int data) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* newNode = (struct LinkedListNode*)malloc(sizeof(struct
LinkedListNode));
    newNode->key = key;
    newNode->data = data;
    newNode->next = NULL;
    if (hashArraySC[hashIndex] == NULL) {
        hashArraySC[hashIndex] = newNode;
    } else {
        struct LinkedListNode* current = hashArraySC[hashIndex];
        struct LinkedListNode* prev = NULL;
        while (current != NULL) {
            if (current->key == key) {
                current->data = data;
                free(newNode);
                return;}
            prev = current;
            current = current->next;}
        prev->next = newNode;}}
struct DataItem* searchLP(int key) {
    int hashIndex = hashCode1(key);
    int originalIndex = hashIndex;
    while (hashArrayLP[hashIndex] != NULL) {
        if (hashArrayLP[hashIndex]->key == key) {
            return hashArrayLP[hashIndex];}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return NULL;}
struct DataItem* searchQP(int key) {
```

```
    int hashIndex = hashCode1(key);
    int i = 0;
    int indexToCheck;
    do {
        indexToCheck = (hashIndex + i*i) % TABLE_SIZE;
        if (hashArrayQP[indexToCheck] == NULL) {
            return NULL;}
        if (hashArrayQP[indexToCheck]->key == key) {
            return hashArrayQP[indexToCheck];}
        i++;
    } while (i < TABLE_SIZE);
    return NULL;}
struct DataItem* searchDH(int key) {
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    int originalIndex = hashIndex;
    while (hashArrayDH[hashIndex] != NULL) {
        if (hashArrayDH[hashIndex]->key == key) {
            return hashArrayDH[hashIndex];}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return NULL;}
struct LinkedListNode* searchSC(int key) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* current = hashArraySC[hashIndex];
    while (current != NULL) {
        if (current->key == key) {
            return current;}
        current = current->next;}
    return NULL;}
bool deleteLP(int key) {
    int hashIndex = hashCode1(key);
    int originalIndex = hashIndex;
    while (hashArrayLP[hashIndex] != NULL) {
        if (hashArrayLP[hashIndex]->key == key) {
            free(hashArrayLP[hashIndex]);
            hashArrayLP[hashIndex] = DELETED_NODE;
            return true;}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return false;}
bool deleteQP(int key) {
    int hashIndex = hashCode1(key);
    int i = 0;
    int indexToCheck;
    do {
        indexToCheck = (hashIndex + i*i) % TABLE_SIZE;
        if (hashArrayQP[indexToCheck] == NULL) {
            return false;}
```

```
        if (hashArrayQP[indexToCheck]->key == key) {
            free(hashArrayQP[indexToCheck]);
            hashArrayQP[indexToCheck] = DELETED_NODE;
            return true;}
        i++;} while (i < TABLE_SIZE);
    return false;}
bool deleteDH(int key) {
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    int originalIndex = hashIndex;
    while (hashArrayDH[hashIndex] != NULL) {
        if (hashArrayDH[hashIndex]->key == key) {
            free(hashArrayDH[hashIndex]);
            hashArrayDH[hashIndex] = DELETED_NODE;
            return true;}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return false;}
bool deleteSC(int key) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* current = hashArraySC[hashIndex];
    struct LinkedListNode* prev = NULL;
    if (current == NULL) {
        return false;}
    if (current->key == key) {
        hashArraySC[hashIndex] = current->next;
        free(current);
        return true;}
    while (current != NULL && current->key != key) {
        prev = current;
        current = current->next;}
    if (current == NULL) {
        return false;}
    prev->next = current->next;
    free(current);
    return true;}
void displayLP() {
    printf("\nLinear Probing Hash Table:\n");
    printf("Index\tKey\tValue\n");
    printf("---------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayLP[i] != NULL && hashArrayLP[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayLP[i]->key, hashArrayLP[i]->data);
        } else if (hashArrayLP[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displayQP() {
    printf("\nQuadratic Probing Hash Table:\n");
    printf("Index\tKey\tValue\n");
```

```
    printf("--------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayQP[i] != NULL && hashArrayQP[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayQP[i]->key, hashArrayQP[i]->data);
        } else if (hashArrayQP[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displayDH() {
    printf("\nDouble Hashing Hash Table:\n");
    printf("Index\tKey\tValue\n");
    printf("--------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayDH[i] != NULL && hashArrayDH[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayDH[i]->key, hashArrayDH[i]->data);
        } else if (hashArrayDH[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displaySC() {
    printf("\nSeparate Chaining Hash Table:\n");
    printf("Index\tEntries\n");
    printf("----------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d\t", i);
        if (hashArraySC[i] == NULL) {
            printf("<empty>");
        } else {
            struct LinkedListNode* current = hashArraySC[i];
            while (current != NULL) {
                printf("[%d,%d]", current->key, current->data);
                if (current->next != NULL) {
                    printf(" -> ");}
                current = current->next;}}
        printf("\n");}}
void cleanupHashTables() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayLP[i] != NULL && hashArrayLP[i] != DELETED_NODE) {
            free(hashArrayLP[i]);}
        if (hashArrayQP[i] != NULL && hashArrayQP[i] != DELETED_NODE) {
            free(hashArrayQP[i]);}
        if (hashArrayDH[i] != NULL && hashArrayDH[i] != DELETED_NODE) {
            free(hashArrayDH[i]);}
        struct LinkedListNode* current = hashArraySC[i];
        while (current != NULL) {
            struct LinkedListNode* temp = current;
            current = current->next;
            free(temp);}}}
void insertInAllTables(int key, int data) {
    insertLP(key, data);
    insertQP(key, data);
```

```
    insertDH(key, data);
    insertSC(key, data);}
void displayMenu() {
    printf("\n===== HASH TABLE DEMONSTRATION =====\n");
    printf("1. Insert a key-value pair");}
int main() {
    int choice, key, data;
    struct DataItem* item;
    struct LinkedListNode* node;
    bool deleted;
    initializeHashTables();
    insertInAllTables(1, 20);
    insertInAllTables(11, 30);
    insertInAllTables(21, 40);
    insertInAllTables(31, 50);
    while (1) {
        displayMenu();
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key and value to insert: ");
                scanf("%d %d", &key, &data);
                insertInAllTables(key, data);
                printf("Inserted [%d, %d] in all tables\n", key, data);
                break;
            case 2:
                printf("Enter key to search: ");
                scanf("%d", &key);
                item = searchLP(key);
                if (item != NULL) {
                    printf("Linear Probing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Linear Probing: Key %d not found\n", key);}
                item = searchQP(key);
                if (item != NULL) {
                    printf("Quadratic Probing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Quadratic Probing: Key %d not found\n", key);}
                item = searchDH(key);
                if (item != NULL) {
                    printf("Double Hashing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Double Hashing: Key %d not found\n", key);}
                node = searchSC(key);
                if (node != NULL) {
                    printf("Separate Chaining: Found [%d, %d]\n", node->key, node-
>data);
                } else {
```

```
                    printf("Separate Chaining: Key %d not found\n", key);}
                break;
            case 3:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                deleted = deleteLP(key);
                printf("Linear Probing: %s\n", deleted ? "Deleted successfully" :
"Key not found");
                deleted = deleteQP(key);
                printf("Quadratic Probing: %s\n", deleted ? "Deleted successfully"
: "Key not found");
                deleted = deleteDH(key);
                printf("Double Hashing: %s\n", deleted ? "Deleted successfully" :
"Key not found");
                deleted = deleteSC(key);
                printf("Separate Chaining: %s\n", deleted ? "Deleted successfully"
: "Key not found");
                break;
            case 4:
                displayLP();
                displayQP();
                displayDH();
                displaySC();
                break;
            case 5:
                cleanupHashTables();
                printf("Exiting program. Memory cleaned up.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");}}
                getch();
    return 0;}
```

OUTPUT:

```
===== HASH TABLE DEMONSTRATION =====
1. Insert a key-value pair1
Enter key and value to insert: 5
1
Inserted [5, 1] in all tables

===== HASH TABLE DEMONSTRATION =====
1. Insert a key-value pair
```

PRACTICAL-19

QUES: *Write a program to implement algorithm for Minimum Spanning Tree.*
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 100
#define INF INT_MAX
typedef struct {
    int src, dest, weight;
} Edge;
typedef struct {
    int V, E;
    Edge* edges;
} Graph;
typedef struct {
    int parent;
    int rank;
} Subset;
Graph* createGraph(int V, int E) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (Edge*)malloc(E * sizeof(Edge));
    return graph;}
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;}
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;}}
int compareEdges(const void* a, const void* b) {
    return ((Edge*)a)->weight - ((Edge*)b)->weight;}
void kruskalMST(Graph* graph) {
    int V = graph->V;
    Edge result[V-1];
    int e = 0;
    int i = 0;
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);
    Subset* subsets = (Subset*)malloc(V * sizeof(Subset));
```

```
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;}
    while (e < V - 1 && i < graph->E) {
        Edge next_edge = graph->edges[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);}}
    printf("\nKruskal's MST:\n");
    int totalWeight = 0;
    for (i = 0; i < e; i++) {
        printf("Edge: %d -- %d  Weight: %d\n",
               result[i].src, result[i].dest, result[i].weight);
        totalWeight += result[i].weight;}
    printf("Total Weight of MST: %d\n", totalWeight);
    free(subsets);}
int minKey(int key[], bool mstSet[], int V) {
    int min = INF, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;}
void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INF, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];}
    printf("\nPrim's MST:\n");
    int totalWeight = 0;
    for (int i = 1; i < V; i++) {
        printf("Edge: %d -- %d  Weight: %d\n", parent[i], i, graph[parent[i]][i]);
        totalWeight += graph[parent[i]][i];}
    printf("Total Weight of MST: %d\n", totalWeight);}
int main() {
    int choice, V, E, u, v, w;
    Graph* graph;
    int adjMatrix[MAX_VERTICES][MAX_VERTICES];
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    for (int i = 0; i < V; i++)
```

```
    for (int j = 0; j < V; j++)
        adjMatrix[i][j] = 0;
printf("Enter number of edges: ");
scanf("%d", &E);
graph = createGraph(V, E);
printf("Enter edge information (source destination weight):\n");
for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    graph->edges[i].src = u;
    graph->edges[i].dest = v;
    graph->edges[i].weight = w;
    adjMatrix[u][v] = w;
    adjMatrix[v][u] = w;}
while (1) {
    printf("\n=== MINIMUM SPANNING TREE ALGORITHMS ===\n");
    printf("1. Find MST using Kruskal's Algorithm\n");
    printf("2. Find MST using Prim's Algorithm\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            kruskalMST(graph);
            break;
        case 2:
            primMST(adjMatrix, V);
            break;
        case 3:
            free(graph->edges);
            free(graph);
            return 0;
        default:
            printf("Invalid choice! Try again.\n");}}
            getch();}
```

PRACTICAL – 20

QUES: *Write a program to implement shortest path algorithm.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 100
#define INF INT_MAX
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int V) {
    int dist[V];
    bool sptSet[V];
    int parent[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = false;
        parent[i] = -1;}
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int min = INF, min_index;
        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
                min = dist[v], min_index = v;
        int u = min_index;
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {
                parent[v] = u;
                dist[v] = dist[u] + graph[u][v];}}
    printf("\nVertex\tDistance\tPath");
    for (int i = 0; i < V; i++) {
        printf("\n%d -> %d\t%d\t\t", src, i, dist[i]);
        int j = i;
        while (j != src) {
            printf("%d <- ", j);
            j = parent[j];}
        printf("%d", src);}
    printf("\n");}
void bellmanFord(int graph[MAX_VERTICES][MAX_VERTICES], int src, int V) {
    int dist[V];
    int parent[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        parent[i] = -1;}
    dist[src] = 0;
    for (int i = 0; i < V - 1; i++) {
        for (int u = 0; u < V; u++) {
            for (int v = 0; v < V; v++) {
```

```
                if (graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] <
dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                    parent[v] = u;}}}}
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
                printf("Graph contains negative weight cycle\n");
                return;}}}
    printf("\nVertex\tDistance\tPath");
    for (int i = 0; i < V; i++) {
        printf("\n%d -> %d\t%d\t\t", src, i, dist[i]);
        int j = i;
        while (j != src && j != -1) {
            printf("%d <- ", j);
            j = parent[j];}
        if (j != -1) printf("%d", src);}
    printf("\n");}
void floydWarshall(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
    int dist[V][V];
    int next[V][V];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
            if (graph[i][j] != 0 && graph[i][j] != INF)
                next[i][j] = j;
            else
                next[i][j] = -1;}}
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];}}}}
    printf("\nAll-Pairs Shortest Paths:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (i != j) {
                printf("\n%d -> %d: Distance = %d, Path: %d", i, j, dist[i][j], i);
                int path = i;
                while (path != j) {
                    path = next[path][j];
                    if (path == -1) {
                        printf(" -> No path exists");
                        break;}
                    printf(" -> %d", path);}}}}
    printf("\n");}
int main() {
    int V, E, choice, src;
    int graph[MAX_VERTICES][MAX_VERTICES];
```

```
printf("Enter number of vertices: ");
scanf("%d", &V);
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        graph[i][j] = 0;}}
printf("Enter number of edges: ");
scanf("%d", &E);
printf("Enter edge information (source destination weight):\n");
for (int i = 0; i < E; i++) {
    int u, v, w;
    scanf("%d %d %d", &u, &v, &w);
    graph[u][v] = w;}
while (1) {
    printf("\n=== SHORTEST PATH ALGORITHMS ===\n");
    printf("1. Dijkstra's Algorithm\n");
    printf("2. Bellman-Ford Algorithm\n");
    printf("3. Floyd-Warshall Algorithm\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Enter source vertex: ");
            scanf("%d", &src);
            dijkstra(graph, src, V);
            break;
        case 2:
            printf("Enter source vertex: ");
            scanf("%d", &src);
            bellmanFord(graph, src, V);
            break;
        case 3:
            floydWarshall(graph, V);
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice! Try again.\n");}}
            getch();}
```