DSA PROGRAMS:

PUSH POP DISPLAY

```
#include <iostream>
using namespace std;
int stack[100], top = -1, n = 100;
void push(int value) {
   if(top >= n - 1)
      cout << "Stack Overflow\n";
   else {
      top++;
      stack[top] = value;
      cout << value << " pushed into stack\n";
   }
}
void pop() {
   if(top < 0)
      cout << "Stack Underflow\n";
   else {
      cout << stack[top] << " popped from stack\n";
      top--;
   }
}
void display() {
   if(top < 0)
      cout << "Stack is empty\n";
   else {
      cout << "Stack elements: ";
      for(int i = top; i >= 0; i--)
         cout << stack[i] << " ";
```

```
        cout << endl;
    }
}

int main() {
    int choice, value;
    while(1) {
        cout << "\n1.Push\n2.Pop\n3.Display\n4.Exit\n";
        cout << "Enter choice: ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "Enter value to push: ";
                cin >> value;
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                cout << "Exiting program...\n";
                return 0;
            default:
                cout << "Invalid choice\n";
        }
    }
}
```

1. Push(item):

Check if stack is full → top == size - 1

If not, increment top and add item at stack[top]

2. Pop():

Check if stack is empty → top == -1

If not, remove stack[top] and decrement top

3. Display():

Traverse from top to 0 and print each element

ENQUEUE DEQUEUE DISPLAY

```
#include <iostream>

using namespace std;

int queue[100];

int front = -1, rear = -1;

void enqueue(int value) {

  if (rear == 99) {

    cout << "Queue is full\n";

  } else {

    if (front == -1) front = 0;

    rear++;

    queue[rear] = value;

    cout << value << " added to queue\n";

  }

}

void dequeue() {

  if (front == -1 || front > rear) {

    cout << "Queue is empty\n";

  } else {

    cout << queue[front] << " removed from queue\n";

    front++;
```

```
    }
}
void display() {
    if (front == -1 || front > rear) {
        cout << "Queue is empty\n";
    } else {
        cout << "Queue: ";
        for (int i = front; i <= rear; i++) {
            cout << queue[i] << " ";
        }
        cout << "\n";
    }
}
int main() {
    int choice, value;

    while (true) {
        cout << "\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter value to enqueue: ";
                cin >> value;
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
```

```
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Invalid choice\n";
        }
    }
    return 0;
}
```

1. Enqueue(item):

Check if queue is full → rear == size - 1

Increment rear, insert item at queue[rear]

2. Dequeue():

Check if queue is empty → front > rear

Increment front

3. Display():

Traverse from front to rear and print each element

SINGLE LINKED LIST(CREATE INSERT DELETE DISPLAY)

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
Node* head = NULL;
// Create/Insert at end
void insertAtEnd(int value) {
    Node* newNode = new Node();
    newNode->data = value;
```

```
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    cout << value << " inserted at end.\n";
}
// Insert at beginning
void insertAtBeginning(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    cout << value << " inserted at beginning.\n";
}

// Delete a node by value
void deleteNode(int value) {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;
    Node* prev = NULL;
    // If head needs to be deleted
    if (head->data == value) {
        head = head->next;
```

```
        delete temp;
        cout << value << " deleted from list.\n";
        return;
    }
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        cout << value << " not found in the list.\n";
        return;
    }
    prev->next = temp->next;
    delete temp;
    cout << value << " deleted from list.\n";
}
// Display the list
void displayList() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;
    cout << "Linked List: ";
    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
// Main with menu
```

```
int main() {
    int choice, value;
    while (true) {
        cout << "\n1. Insert at End\n2. Insert at Beginning\n3. Delete Node\n4. Display List\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter value: ";
                cin >> value;
                insertAtEnd(value);
                break;
            case 2:
                cout << "Enter value: ";
                cin >> value;
                insertAtBeginning(value);
                break;
            case 3:
                cout << "Enter value to delete: ";
                cin >> value;
                deleteNode(value);
                break;
            case 4:
                displayList();
                break;
            case 5:
                cout << "Exiting...\n";
                return 0;
            default:
                cout << "Invalid choice.\n";
        }
```

```
    }

    return 0;
}
```

1. CreateNode(data):

Allocate node, set node->data = data, node->next = NULL

If head is NULL, make head point to node

Else, traverse to end and insert

2. InsertAtPosition(pos, data):

Traverse to pos-1, insert new node between nodes

3. DeleteAtPosition(pos):

Traverse to pos-1, delete pos node by adjusting links

4. Display():

Traverse from head to NULL and print data

SINGLE LINKED LIST(CREATE INSERT SEARCH DISPLAY)

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
Node* head = NULL;
// Create/Insert at end
void insertAtEnd(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
```

```
        Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    cout << value << " inserted into the list.\n";
}
// Search a value
void search(int key) {
    Node* temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == key) {
            cout << key << " found at position " << pos << ".\n";
            return;
        }
        temp = temp->next;
        pos++;
    }
    cout << key << " not found in the list.\n";
}
// Display the list
void display() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;
    cout << "List: ";
    while (temp != NULL) {
        cout << temp->data << " ";
```

```
        temp = temp->next;
    }
    cout << endl;
}
// Main function with menu
int main() {
    int choice, value;
    while (true) {
        cout << "\n1. Insert (Create)\n2. Search\n3. Display\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter value to insert: ";
                cin >> value;
                insertAtEnd(value);
                break;
            case 2:
                cout << "Enter value to search: ";
                cin >> value;
                search(value);
                break;
            case 3:
                display();
                break;
            case 4:
                cout << "Exiting...\n";
                return 0;
            default:
                cout << "Invalid choice.\n";
        }
```

```
    }
}
```

1. ***CreateNode(data)***
   Allocate node, set node->data = data, node->next = NULL
   If head is NULL, make head point to node
   Else, traverse to end and insert

2. Search(data):

Traverse and compare node->data with search value

3. Display() :

Traverse from head to NULL and print data

BUBBLE SORT ALGORITHM

```
#include <iostream>
using namespace std;
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Last i elements are already sorted
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap if the element is greater than the next
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
void displayArray(int arr[], int n) {
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}
int main() {
    int n, arr[100];
    cout << "Enter number of elements: ";
    cin >> n;
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    bubbleSort(arr, n);
    displayArray(arr, n);
 return 0;
}
```

1. Loop i from 0 to n-1

2. Loop j from 0 to n-i-1

If arr[j] > arr[j+1], swap them

3. Repeat until no swaps in inner loop

TREE TRAVERSAL(INORDER PREORDER POSTORDER)

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
};
// Create new node
Node* createNode(int value) {
```

```
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}
// Insert node into BST
Node* insert(Node* root, int value) {
    if (root == NULL)
        return createNode(value);
    if (value < root->data)
        root->left = insert(root->left, value);
    else
        root->right = insert(root->right, value);
    return root;
}
// Inorder traversal
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
// Preorder traversal
void preorder(Node* root) {
    if (root != NULL) {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
// Postorder traversal
void postorder(Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << " ";
    }
}
int main() {
    Node* root = NULL;
    int n, value;
    cout << "Enter number of nodes to insert: ";
    cin >> n;
    cout << "Enter " << n << " values:\n";
    for (int i = 0; i < n; i++) {
        cin >> value;
        root = insert(root, value);
    }
    cout << "\nInorder Traversal: ";
    inorder(root);
    cout << "\nPreorder Traversal: ";
    preorder(root);
    cout << "\nPostorder Traversal: ";
    postorder(root);
    cout << endl;
    return 0;
}
```

1. Inorder(root):

Traverse left, visit root, traverse right

2. Preorder(root):

Visit root, traverse left, traverse right

3. Postorder(root):

Traverse left, traverse right, visit root

LINEAR SEARCH & BINARY SEARCH

```
#include <iostream>

#include <algorithm> // for sort()

using namespace std;


// Linear Search

bool linearSearch(int arr[], int n, int key) {

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

      if (arr[i] == key)

         return true;

   }

   return false;

}


// Binary Search (array must be sorted)

bool binarySearch(int arr[], int n, int key) {

   int low = 0, high = n - 1;

   while (low <= high) {

      int mid = (low + high) / 2;

      if (arr[mid] == key)

         return true;

      else if (arr[mid] < key)

         low = mid + 1;

      else

         high = mid - 1;
```

```
    }
    return false;
}

int main() {
    int arr[100], n, key;

    cout << "Enter number of elements: ";
    cin >> n;

    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter value to search: ";
    cin >> key;

    // Linear Search
    if (linearSearch(arr, n, key))
        cout << "Linear Search: Element found!\n";
    else
        cout << "Linear Search: Element not found.\n";

    // Sort for Binary Search
    sort(arr, arr + n);

    // Binary Search
    if (binarySearch(arr, n, key))
        cout << "Binary Search: Element found!\n";
    else
        cout << "Binary Search: Element not found.\n";
```

```
    return 0;
}
```

Linear Search Algorithm:

Traverse array

If arr[i] == key, return index

Binary Search Algorithm (sorted array):

1. Set low = 0, high = n-1

2. While low <= high

mid = (low + high)/2

If arr[mid] == key, return mid

If key < arr[mid], set high = mid - 1

Else, set low = mid + 1

ADJACENCY MATRIX IMPLEMENTATION

```
#include <iostream>
using namespace std;
class Graph {
private:
    int **adjMatrix; // Pointer to adjacency matrix
    int numVertices; // Number of vertices
public:
    // Constructor to initialize graph
    Graph(int vertices) {
        numVertices = vertices;
        adjMatrix = new int*[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjMatrix[i] = new int[numVertices];
```

```
        }
        // Initialize matrix to 0
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                adjMatrix[i][j] = 0;
            }
        }
    }
    // Add edge to the graph
    void addEdge(int startVertex, int endVertex) {
        adjMatrix[startVertex][endVertex] = 1;
        adjMatrix[endVertex][startVertex] = 1; // For undirected graph
    }
    // Display the adjacency matrix
    void displayMatrix() {
        cout << "Adjacency Matrix:\n";
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
    // Destructor to free memory
    ~Graph() {
        for (int i = 0; i < numVertices; i++) {
            delete[] adjMatrix[i];
        }
        delete[] adjMatrix;
    }
};
```

```
int main() {
    int vertices, edges, start, end;
    cout << "Enter number of vertices: ";
    cin >> vertices;
    Graph g(vertices);
    cout << "Enter number of edges: ";
    cin >> edges;
    for (int i = 0; i < edges; i++) {
        cout << "Enter edge (startVertex endVertex): ";
        cin >> start >> end;
        g.addEdge(start, end);
    }
    g.displayMatrix();
    return 0;
}
```

Algorithm:

1. Initialize 2D array adj[n][n] with 0

2. For each edge (u, v):

Set adj[u][v] = 1

If undirected, also set adj[v][u] = 1

HASH TABLE IMPLEMENTATION

```
#include <iostream>
#include <list>
using namespace std;

// Hash Table class
class HashTable {
private:
    static const int tableSize = 10; // Size of hash table
```

```
    list<int> *table;  // Array of linked lists

public:
    // Constructor
    HashTable() {
        table = new list<int>[tableSize];  // Create an array of lists
    }

    // Hash function
    int hashFunction(int key) {
        return key % tableSize;  // Simple hash function (modulo)
    }

    // Insert a key into the hash table
    void insert(int key) {
        int index = hashFunction(key);
        table[index].push_back(key);  // Insert the key at the appropriate index
    }

    // Search for a key in the hash table
    bool search(int key) {
        int index = hashFunction(key);
        for (int x : table[index]) {
            if (x == key) {
                return true;  // Found the key
            }
        }
        return false;  // Key not found
    }

    // Delete a key from the hash table
```

```
    void deleteKey(int key) {
        int index = hashFunction(key);
        table[index].remove(key);  // Remove the key from the list
    }

    // Display the hash table
    void display() {
        for (int i = 0; i < tableSize; i++) {
            cout << "Index " << i << ": ";
            for (int x : table[i]) {
                cout << x << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    HashTable ht;
    int choice, key;

    while (true) {
        cout << "\nMenu:\n";
        cout << "1. Insert a key\n";
        cout << "2. Search for a key\n";
        cout << "3. Delete a key\n";
        cout << "4. Display hash table\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
```

```
switch (choice) {
    case 1:
        cout << "Enter the key to insert: ";
        cin >> key;
        ht.insert(key);
        cout << "Key inserted successfully.\n";
        break;

    case 2:
        cout << "Enter the key to search: ";
        cin >> key;
        if (ht.search(key)) {
            cout << "Key " << key << " found in the hash table.\n";
        } else {
            cout << "Key " << key << " not found in the hash table.\n";
        }
        break;

    case 3:
        cout << "Enter the key to delete: ";
        cin >> key;
        ht.deleteKey(key);
        cout << "Key " << key << " deleted successfully.\n";
        break;

    case 4:
        cout << "Hash Table contents:\n";
        ht.display();
        break;

    case 5:
```

```
            cout << "Exiting program.\n";
            return 0;

          default:
            cout << "Invalid choice! Please try again.\n";
        }
    }

    return 0;
}
```

Algorithm (using Chaining):

1. Initialize an array of linked lists

2. Insert(key):

Compute hash index = key % size

Insert key in the linked list at hash[index]

3. Search(key):

Compute hash index, search in the list

4. Delete(key):

Locate and delete node from list at hash index

DIJKSHTRA ALGORITHM

```
#include <iostream>
#include <climits>
using namespace std;

#define V 4

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (!sptSet[v] && dist[v] <= min)
```

```
            min = dist[v], min_index = v;
    return min_index;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    cout << "Vertex Distance from Source\n";
    for (int i = 0; i < V; i++)
        cout << i << "\t" << dist[i] << endl;
}

int main() {
    int graph[V][V] = {{0, 4, 8, 0},
                       {4, 0, 2, 5},
                       {8, 2, 0, 3},
                       {0, 5, 3, 0}};
    dijkstra(graph, 0);
    return 0;
}
```

1. Initialize distance of source to 0 and all others to INF

2. Use visited[] to track processed nodes

3. For each unvisited node with smallest distance:

Mark it visited

Update its neighbors if a shorter path is found via this node