SECTION : A

Q1 . 20 friends put their wallets in a row. The first wallet contains 20 dollars, the second has 30 dollars, the third has 40 dollars, and so on, with each wallet having 10 dollars more than the previous one. Since the data is already sorted in ascending order, no sorting is required. But if you are given a chance to sort the wallets, which sorting technique would be best to apply? Write a C++ program to implement your chosen sorting approach.

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

// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];     // Current element
        int j = i - 1;

        // Shift elements that are greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;  // Place key at correct position
    }
```

```
}

int main() {
    int n = 20;
    int wallets[n];

    // Initializing wallets (20, 30, 40, ..., 210)
    for (int i = 0; i < n; i++) {
        wallets[i] = 20 + (i * 10);
    }

    cout << "Original Wallet Amounts (Already Sorted):" << endl;
    for (int i = 0; i < n; i++) {
        cout << wallets[i] << " ";
    }
    cout << endl;

    // Sorting (though already sorted)
    insertionSort(wallets, n);

    cout << "\nAfter Applying Insertion Sort:" << endl;
    for (int i = 0; i < n; i++) {
        cout << wallets[i] << " ";
    }
    cout << endl;

    return 0;
```

}

The best sorting technique to apply is Insertion Sort, because it works in O(n) time when the array is already sorted. The above C++ program demonstrates this approach.

Q 2: In a park, 10 friends were discussing a game based on sorting. They placed their wallets in a row. The maximum money in any wallet is $6. Among them, 3 wallets contain exactly $2, 2 wallets contain exactly $3, 2 wallets are empty ($0), 1 wallet contains $1, and 1 wallet contains $4. Which sorting technique would you apply to sort the wallets on the basis of the money they contain? Write a program to implement your chosen sorting technique.

Why Counting Sort?

- Wallets contain money in range 0–6 (small range).
- Number of wallets = 10.
- Counting Sort is best because it works in O(n + k) time, which is efficient compared to O(n^2) methods like bubble sort.

```
#include <iostream>
#include <vector>
#include <algorithm> // for max_element
using namespace std;

void countingSort(vector<int>& wallets) {
    // Step 1: Find maximum value
    int maxVal = *max_element(wallets.begin(), wallets.end());
```

```
    // Step 2: Create count array
    vector<int> count(maxVal + 1, 0);

    // Step 3: Count frequencies
    for (int money : wallets) {
        count[money]++;
    }

    // Step 4: Reconstruct sorted array
    int index = 0;
    for (int value = 0; value <= maxVal; value++) {
        while (count[value] > 0) {
            wallets[index++] = value;
            count[value]--;
        }
    }
}

int main() {
    // Example data (wallets amounts)
    vector<int> wallets = {2, 3, 0, 2, 4, 1, 3, 2, 0, 6};

    cout << "Original wallets: ";
    for (int money : wallets) cout << money << " ";
    cout << endl;

    // Sort wallets
```

```
    countingSort(wallets);

    cout << "Sorted wallets:   ";
    for (int money : wallets) cout << money << " ";
    cout << endl;

    return 0;
}
```

Original wallets: 2 3 0 2 4 1 3 2 0 6

Sorted wallets: 0 0 1 2 2 2 3 3 4 6

Q 3: During a college fest, 12 students participated in a gaming competition. Each student's score was recorded as follows: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76 The organizers want to arrange the scores in ascending order to decide the ranking of the players. Since the data set is unsorted and contains numbers spread across a wide range, the most efficient technique to apply here is Quick Sort. Write a C++ program to implement Quick Sort to arrange the scores in ascending order.

Why Quick Sort?

- Average case time complexity: O(n log n)
- Faster than Bubble Sort, Insertion Sort, or Selection Sort ($O(n^2)$).
- Works well for datasets with large value ranges.

```
#include <iostream>
```

```
using namespace std;

// Function to swap two elements
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // choose last element as pivot
    int i = low - 1; // index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
```

```
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int scores[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76};
    int n = sizeof(scores) / sizeof(scores[0]);

    cout << "Original scores: ";
    for (int i = 0; i < n; i++) cout << scores[i] << " ";
    cout << endl;

    quickSort(scores, 0, n - 1);

    cout << "Sorted scores (ascending): ";
    for (int i = 0; i < n; i++) cout << scores[i] << " ";
    cout << endl;

    return 0;
}
```

```
Original scores: 45 12 78 34 23 89 67 11 90 54 32 76
```

Sorted scores (ascending): 11 12 23 32 34 45 54 67 76 78 89 90

✅ Final Answer:

Sorting technique: Quick Sort

Time Complexity:

Best/Average case: O(n log n)

Worst case: $O(n^2)$ (when pivot is poorly chosen)

Space Complexity: O(log n) (for recursion stack)

Program provided in C++ above.

Q 4: A software company is tracking project deadlines (in days remaining to submit). The deadlines are: 25, 12, 45, 7, 30, 18, 40, 22, 10, 35. The manager wants to arrange the deadlines in ascending order to prioritize the projects with the least remaining time. For efficiency, the project manager hints to the team to apply a divide-and-conquer technique that divides the array into unequal parts. Write a C++ program to sort the project deadlines using the above sorting technique.

Why Quick Sort?

- Quick Sort is a **divide-and-conquer algorithm**.
- Unlike Merge Sort (which always divides into equal halves), Quick Sort divides arrays into **unequal partitions** based on a pivot.
- Efficient for medium-size datasets like this.

```
#include <iostream>
```

```
using namespace std;

// Function to swap two elements
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
```

```
        int pi = partition(arr, low, high);

        // Recursively sort sub-arrays
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int deadlines[] = {25, 12, 45, 7, 30, 18, 40, 22, 10, 35};
    int n = sizeof(deadlines) / sizeof(deadlines[0]);

    cout << "Original deadlines: ";
    for (int i = 0; i < n; i++) cout << deadlines[i] << " ";
    cout << endl;

    quickSort(deadlines, 0, n - 1);

    cout << "Sorted deadlines (ascending): ";
    for (int i = 0; i < n; i++) cout << deadlines[i] << " ";
    cout << endl;

    return 0;
}
```

```
Original deadlines: 25 12 45 7 30 18 40 22 10 35
```

Sorted deadlines (ascending): 7 10 12 18 22 25 30 35 40 45

✅ Final Answer

Sorting Technique: Quick Sort (divide-and-conquer with unequal partitions).

Time Complexity: O(n log n) average, O(n^2) worst case.

Space Complexity: O(log n).

C++ code provided above.

Q 5: Suppose there is a square named SQ-1. By connecting the midpoints of SQ-1, we create another square named SQ-2. Repeating this process, we create a total of 50 squares {SQ-1, SQ-2, …, SQ-50}. The areas of these squares are stored in an array. Your task is to search whether a given area is present in the array or not.What would be the best searching approach? Write a C++ program to implement this approach.

Choosing the Searching Technique

- The areas are stored in an array.
- Since each next square's area is half the previous one, the array is sorted in descending order.

Best searching approach: Binary Search ✅

- Works efficiently on sorted arrays.
- Time Complexity: O(log n)
- Better than Linear Search (O(n)).

#include <iostream>

```
#include <cmath>
using namespace std;

// Binary Search function
bool binarySearch(double arr[], int n, double key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        if (fabs(arr[mid] - key) < 1e-6) // handle floating point precision
            return true;
        else if (arr[mid] > key)
            low = mid + 1; // search right side (since descending order)
        else
            high = mid - 1; // search left side
    }
    return false;
}

int main() {
    double A; // Area of SQ-1
    cout << "Enter area of SQ-1: ";
    cin >> A;

    double areas[50];
    areas[0] = A;
```

```
    // Fill array with areas of 50 squares
    for (int i = 1; i < 50; i++) {
        areas[i] = areas[i - 1] / 2.0;
    }

    double key;
    cout << "Enter area to search: ";
    cin >> key;

    if (binarySearch(areas, 50, key))
        cout << "Area " << key << " is present in the array." << endl;
    else
        cout << "Area " << key << " is NOT present in the array." << endl;

    return 0;
}
```

Input:

Enter area of SQ-1: 256

Enter area to search: 32

Process:

- SQ-1 = 256
- SQ-2 = 128
- SQ-3 = 64
- SQ-4 = 32 ✅ found

Output:

Area 32 is present in the array.

Q 6: Before a match, the chief guest wants to meet all the players. The head coach introduces the first player, then that player introduces the next player, and so on, until all players are introduced. The chief guest moves forward with each introduction, meeting the players one at a time. How would you implement the above activity using a Linked List? Write a C++ program to implement the logic.

Why Linked List?

- Dynamic structure (players can be added/removed easily).
- Sequential access (one player introduces the next).
- Natural mapping to the problem's description.

Linked List Operations Needed

1. Node creation: Store player name and pointer to next.
2. Insert player: Add new player at the end of the list.
3. Traverse list: Chief guest meets players one by one.

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

// Node structure for each player
struct Player {
    string name;
    Player* next;

    Player(string n) {
        name = n;
```

```
        next = nullptr;
    }
};

// Linked List class
class PlayerList {
private:
    Player* head;

public:
    PlayerList() {
        head = nullptr;
    }

    // Function to add a player at the end
    void addPlayer(string name) {
        Player* newPlayer = new Player(name);
        if (head == nullptr) {
            head = newPlayer;
        } else {
            Player* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newPlayer;
        }
    }
```

```
    // Function to introduce players one by one
    void introducePlayers() {
        if (head == nullptr) {
            cout << "No players available." << endl;
            return;
        }

        cout << "Chief Guest meets the players:" << endl;
        Player* temp = head;
        int count = 1;
        while (temp != nullptr) {
            cout << "Meeting Player " << count << ": " << temp->name << endl;
            temp = temp->next;
            count++;
        }
    }
};

int main() {
    PlayerList team;

    // Add players
    team.addPlayer("Rohit");
    team.addPlayer("Virat");
    team.addPlayer("Rahul");
    team.addPlayer("Jadeja");
```

```
    team.addPlayer("Bumrah");

    // Chief guest meets players
    team.introducePlayers();

    return 0;
}
```

Chief Guest meets the players:
Meeting Player 1: Rohit
Meeting Player 2: Virat
Meeting Player 3: Rahul
Meeting Player 4: Jadeja
Meeting Player 5: Bumrah

✅ Final Answer

Data structure: Singly Linked List.

Each player = Node.

Chief guest meeting players = Traversing the linked list.

C++ program provided above.

Q 7: A college bus travels from stop A → stop B → stop C → stop D and then returns in reverse order D → C → B →

A. Model this journey using a doubly linked list. Write a program to:

- **Store bus stops in a doubly linked list.**
- **Traverse forward to show the onward journey.**
- **Traverse backward to show the return journey.**

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

// Node structure for doubly linked list
struct Node {
    string data;
    Node* prev;
    Node* next;
};

// Function to create a new node
Node* createNode(string data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->prev = nullptr;
    newNode->next = nullptr;
    return newNode;
}

// Function to append a node at the end
```

```
void append(Node*& head, string data) {
    Node* newNode = createNode(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to traverse forward
Node* traverseForward(Node* head) {
    Node* temp = head;
    Node* last = nullptr;
    cout << "Onward journey: ";
    while (temp != nullptr) {
        cout << temp->data;
        if (temp->next != nullptr) cout << " -> ";
        last = temp;
        temp = temp->next;
    }
    cout << endl;
    return last; // return last node for backward traversal
```

```
}

// Function to traverse backward
void traverseBackward(Node* last) {
    Node* temp = last;
    cout << "Return journey: ";
    while (temp != nullptr) {
        cout << temp->data;
        if (temp->prev != nullptr) cout << " -> ";
        temp = temp->prev;
    }
    cout << endl;
}

// Main function
int main() {
    Node* head = nullptr;

    // Add bus stops
    string stops[] = {"A", "B", "C", "D"};
    for (string stop : stops) {
        append(head, stop);
    }

    // Traverse forward
    Node* lastNode = traverseForward(head);
```

```
    // Traverse backward
    traverseBackward(lastNode);

    return 0;
}
```

Onward journey: A -> B -> C -> D

Return journey: D -> C -> B -> A

Q 8: There are two teams named Dalta Gang and Malta Gang. Dalta Gang has 4 members, and each member has 2 Gullaks (piggy banks) with some money stored in them. Malta Gang has 2 members, and each member has 3 Gullaks. Both gangs store their Gullak money values in a 2D array. Write a C++ program to:

- **Display the stored data in matrix form.**
- **To multiply Dalta Gang matrix with Malta Gang Matrix**

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

int main() {
    // Dalta Gang: 4 members, 2 Gullaks each (4x2 matrix)
```

```
int dalta[4][2] = {
    {10, 20},
    {30, 40},
    {50, 60},
    {70, 80}
};

// Malta Gang: 2 members, 3 Gullaks each (2x3 matrix)
int malta[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};

// Display Dalta Gang matrix
cout << "Dalta Gang matrix (4x2):" << endl;
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 2; j++) {
        cout << dalta[i][j] << "\t";
    }
    cout << endl;
}

cout << endl;

// Display Malta Gang matrix
cout << "Malta Gang matrix (2x3):" << endl;
for (int i = 0; i < 2; i++) {
```

```
    for (int j = 0; j < 3; j++) {
        cout << malta[i][j] << "\t";
    }
    cout << endl;
}

cout << endl;

// Multiply Dalta Gang matrix (4x2) with Malta Gang matrix (2x3)
int result[4][3] = {0}; // Initialize 4x3 result matrix with zeros

for (int i = 0; i < 4; i++) {         // rows of Dalta
    for (int j = 0; j < 3; j++) {     // columns of Malta
        for (int k = 0; k < 2; k++) { // common dimension
            result[i][j] += dalta[i][k] * malta[k][j];
        }
    }
}

// Display result
cout << "Result of multiplication (4x3 matrix):" << endl;
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        cout << result[i][j] << "\t";
    }
    cout << endl;
}
```

```
    return 0;
}
```

- Matrix representation:
  - Dalta Gang:
  - 10 20
  - 30 40
  - 50 60
  - 70 80
  - Malta Gang:
  - 1 2 3
  - 4 5 6
- Matrix multiplication formula:
  - result[i][j] = sum(dalta[i][k] * malta[k][j]) for k=0 to 1
- Resulting matrix (4x3):

90 110 130

190 230 270

290 350 410

390 470 550

SECTION : B

Q 1: To store the names of family members, an expert suggests organizing the data in a way that allows efficient searching, traversal, and insertion of new members. For this purpose, use a Binary Search Tree (BST) to store the names of family members, starting with the letters:

<Q, S, R, T, M, A, B, P, N>

Write a C++ program to Create a Binary Search Tree (BST) using the given names and find and display the successor of the family member whose name starts with M.

BST Node Structure

A BST node will store:

- The name (string).
- Pointers to the left and right child.

- Insertion Rule
- In a BST, left child < parent < right child (lexicographically).

- We'll insert names based on string comparison.

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

// Node structure for BST
struct Node {
    string name;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(string name) {
    Node* newNode = new Node();
    newNode->name = name;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Function to insert a name into BST
Node* insert(Node* root, string name) {
    if (root == nullptr)
        return createNode(name);
    if (name < root->name)
        root->left = insert(root->left, name);
    else
```

```
        root->right = insert(root->right, name);
    return root;
}

// Function to find minimum value node in a subtree
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
}

// Function to find inorder successor
Node* inorderSuccessor(Node* root, Node* target) {
    if (target->right != nullptr)
        return minValueNode(target->right);

    Node* successor = nullptr;
    Node* ancestor = root;
    while (ancestor != nullptr) {
        if (target->name < ancestor->name) {
            successor = ancestor;
            ancestor = ancestor->left;
        } else if (target->name > ancestor->name) {
            ancestor = ancestor->right;
        } else
            break;
```

```
    }
    return successor;
}

// Function to search a node by name
Node* search(Node* root, string name) {
    if (root == nullptr || root->name == name)
        return root;
    if (name < root->name)
        return search(root->left, name);
    else
        return search(root->right, name);
}

// Function for inorder traversal
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->name << " ";
        inorder(root->right);
    }
}

// Main function
int main() {
    Node* root = nullptr;
    string names[] = {"Q", "S", "R", "T", "M", "A", "B", "P", "N"};
```

```
    // Insert names into BST
    for (string name : names) {
        root = insert(root, name);
    }

    cout << "Inorder traversal of BST: ";
    inorder(root);
    cout << endl;

    // Find successor of "M"
    Node* target = search(root, "M");
    if (target != nullptr) {
        Node* successor = inorderSuccessor(root, target);
        if (successor != nullptr)
            cout << "Successor of " << target->name << " is: " << successor->name
<< endl;
        else
            cout << target->name << " has no successor." << endl;
    } else {
        cout << "Name not found in BST." << endl;
    }

    return 0;
}
```

Explanation:

1. BST Construction:
    - Insert names lexicographically: A, B, M, N, P, Q, R, S, T.
2. Inorder Traversal:
    - Prints names in sorted order: A B M N P Q R S T.
3. Finding Successor:
    - Successor of M is the next higher node in inorder traversal → N.

Output:

Inorder traversal of BST: A B M N P Q R S T

Successor of M is: N

Q 2: Implement the In-Order, Pre- Order and Post-Order traversal of Binary search tree with help of C++ Program.

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

// Node structure for BST
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
```

```
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Function to insert a node into BST
Node* insert(Node* root, int data) {
    if (root == nullptr)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}

// In-Order Traversal: Left -> Root -> Right
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
```

```
// Pre-Order Traversal: Root -> Left -> Right
void preorder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
}

// Post-Order Traversal: Left -> Right -> Root
void postorder(Node* root) {
    if (root != nullptr) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << " ";
    }
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into BST
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    for (int val : values) {
        root = insert(root, val);
```

```
    }

    cout << "In-Order Traversal: ";
    inorder(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    preorder(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    postorder(root);
    cout << endl;

    return 0;
}
```

Explanation:

1. Insert Function:
    - Inserts nodes in BST according to BST rules: left < root < right.
2. Traversals:
    - In-Order: 20 30 40 50 60 70 80
    - Pre-Order: 50 30 20 40 70 60 80
    - Post-Order: 20 40 30 60 80 70 50

Q 3: Write a C++ program to search an element in a given binary search Tree.

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Function to insert a node in BST
Node* insert(Node* root, int data) {
    if (root == nullptr)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
```

```
    else
        root->right = insert(root->right, data);
    return root;
}

// Function to search a node in BST
bool search(Node* root, int key) {
    if (root == nullptr)
        return false;
    if (root->data == key)
        return true;
    else if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into BST
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    for (int val : values)
        root = insert(root, val);

    int key;
```

```
    cout << "Enter element to search: ";
    cin >> key;

    if (search(root, key))
        cout << "Element " << key << " found in BST." << endl;
    else
        cout << "Element " << key << " not found in BST." << endl;

    return 0;
}
```

✅ Explanation:

1. BST Creation:
    - Insert nodes using the insert function according to BST rules.
2. Search Function:
    - Recursively compares the key with the current node's value.
    - Traverses left or right subtree accordingly.

Example Run:

Enter element to search: 60

Element 60 found in BST.

Enter element to search: 25

Element 25 not found in BST.

Q 4: In a university, the roll numbers of newly admitted students are: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54

The administration wants to store these roll numbers in a way that allows fast searching, insertion, and retrieval in ascending order. For efficiency, they decide to apply a Binary Search Tree (BST).

Write a C++ program to construct a Binary Search Tree using the above roll numbers and perform an in-order traversal to display them in ascending order.

Step 1: BST Node Structure

Each node contains:

- data (integer for roll number)
- left and right child pointers

Step 2: Insertion Rule

- If the tree is empty → create a new node.
- If value < node's data → insert in left subtree.
- If value > node's data → insert in right subtree.

Step 3: In-Order Traversal

- Left → Root → Right
- For BST, in-order traversal gives values in ascending order.

```
#include <iostream>
```

```
using namespace std;

// Node structure for BST
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Function to insert a node into BST
Node* insert(Node* root, int data) {
    if (root == nullptr)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
```

```
}

// In-order traversal: Left -> Root -> Right
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// Main function
int main() {
    Node* root = nullptr;
    int rollNumbers[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};

    // Construct BST
    for (int roll : rollNumbers) {
        root = insert(root, roll);
    }

    // Display roll numbers in ascending order
    cout << "Roll numbers in ascending order: ";
    inorder(root);
    cout << endl;

    return 0;
```

}

Explanation:

1. BST Construction:
    - Nodes are inserted using the insert function according to BST rules.
2. In-Order Traversal:
    - Prints numbers in ascending order:

11 12 23 34 45 54 67 78 89 90

Q 5: In a university database, student roll numbers are stored using a Binary Search Tree (BST) to allow efficient searching, insertion, and deletion. The roll numbers are: 50, 30, 70, 20, 40, 60, 80. The administrator now wants to delete a student record from the BST. Write a C++ program to delete a node (student roll number) entered by the user.

BST Node Structure

Each node contains:

- data (integer for roll number)
- left and right child pointers

Deletion in BST

There are three cases when deleting a node:

1. Node has no child → simply delete it.
2. Node has one child → delete it and link its child to its parent.

3. Node has two children → find inorder successor (smallest node in right subtree), replace node's value with successor's value, then delete the successor.

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

// Node structure for BST
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert node into BST
Node* insert(Node* root, int data) {
    if (root == nullptr)
        return createNode(data);
```

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

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

// Delete a node in BST
Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    // Search for the node
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node found
```

```
        // Case 1: No child
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            return nullptr;
        }

        // Case 2: One child
        else if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Case 3: Two children
        Node* temp = minValueNode(root->right); // inorder successor
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

```
// Inorder traversal
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// Main function
int main() {
    Node* root = nullptr;
    int rollNumbers[] = {50, 30, 70, 20, 40, 60, 80};

    // Construct BST
    for (int roll : rollNumbers)
        root = insert(root, roll);

    cout << "BST Inorder before deletion: ";
    inorder(root);
    cout << endl;

    int key;
    cout << "Enter roll number to delete: ";
    cin >> key;

    root = deleteNode(root, key);
```

```
    cout << "BST Inorder after deletion: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Explanation:

1. BST Construction:
    - Nodes inserted according to BST rules: 50, 30, 70, 20, 40, 60, 80.
2. Delete Node:
    - Handles all three cases (no child, one child, two children).
3. In-Order Traversal:
    - Prints BST in ascending order before and after deletion.

Example Run:

BST Inorder before deletion: 20 30 40 50 60 70 80

Enter roll number to delete: 30

BST Inorder after deletion: 20 40 50 60 70 80

Q 6: Design and implement a family tree hierarchy using a Binary Search Tree (BST). The family tree should allow efficient storage, retrieval, and manipulation of information related to individuals and their relationships within the family.

Write a C++ program to:

1. **Insert family members into the BST (based on their names).**
2. **Perform in-order, pre-order, and post-order traversals to display the hierarchy.**
3. **Search for a particular family member by name.**

BST Node Structure

Each node will store:

- name (string)
- left and right child pointers

BST property:

- Left child < Parent < Right child (lexicographically by name)

Step 2: Traversals

1. **In-Order (Left → Root → Right)** – Displays names in **alphabetical order**.
2. **Pre-Order (Root → Left → Right)** – Can represent hierarchical insertion order.
3. **Post-Order (Left → Right → Root)** – Useful for deleting nodes or bottom-up traversal.

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

// Node structure for BST
struct Node {
    string name;
    Node* left;
    Node* right;
};

// Create a new node
Node* createNode(string name) {
    Node* newNode = new Node();
    newNode->name = name;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert node into BST based on name
Node* insert(Node* root, string name) {
    if (root == nullptr)
        return createNode(name);
    if (name < root->name)
        root->left = insert(root->left, name);
    else if (name > root->name)
        root->right = insert(root->right, name);
```

```
    // If name already exists, do nothing
    return root;
}

// In-Order Traversal: Left -> Root -> Right
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->name << " ";
        inorder(root->right);
    }
}

// Pre-Order Traversal: Root -> Left -> Right
void preorder(Node* root) {
    if (root != nullptr) {
        cout << root->name << " ";
        preorder(root->left);
        preorder(root->right);
    }
}

// Post-Order Traversal: Left -> Right -> Root
void postorder(Node* root) {
    if (root != nullptr) {
        postorder(root->left);
        postorder(root->right);
```

```
        cout << root->name << " ";
    }
}

// Search for a family member
Node* search(Node* root, string name) {
    if (root == nullptr || root->name == name)
        return root;
    if (name < root->name)
        return search(root->left, name);
    else
        return search(root->right, name);
}

// Main function
int main() {
    Node* root = nullptr;
    string members[] = {"John", "Alice", "Bob", "Diana", "Charles", "Eve"};

    // Insert family members into BST
    for (string name : members)
        root = insert(root, name);

    // Display hierarchy
    cout << "In-Order Traversal (Alphabetical Order): ";
    inorder(root);
    cout << endl;
```

```
    cout << "Pre-Order Traversal: ";
    preorder(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    postorder(root);
    cout << endl;

    // Search for a family member
    string searchName;
    cout << "Enter name to search: ";
    cin >> searchName;

    Node* found = search(root, searchName);
    if (found != nullptr)
        cout << searchName << " is found in the family tree." << endl;
    else
        cout << searchName << " is not found in the family tree." << endl;

    return 0;
}
```

Explanation:

1. **Insertion**:
    - Names are inserted lexicographically to maintain BST property.
2. **Traversals**:
    - **In-Order** → Alphabetical: Alice Bob Charles Diana Eve John

- **Pre-Order** → Root first: John Alice Bob Diana Charles Eve
- **Post-Order** → Root last: Bob Charles Diana Eve Alice John

3. **Search**:
    - Recursively finds the name in the BST.