# DSA LAB QUESTIONS

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.**

For a list that is already sorted or nearly sorted, insertion sort is considered the best technique because it runs in linear time O(n)$O(n)$ on sorted data and requires minimal comparisons and swaps. This is much more efficient than other techniques in this scenario.

**Explanation and Chosen Sorting Technique**

- Insertion sort is optimal for sorted or nearly sorted arrays since it only needs to check if each element is in the correct position, resulting in O(n)$O(n)$ time complexity for a sorted list.
- Algorithms like bubble sort also have linear best-case performance for sorted lists, but insertion sort is preferred for its simplicity and fewer swaps.
- More complex algorithms, such as merge sort or quicksort, would be unnecessary here, as their minimum time complexity is O(nlogn)$O(n\log n)$.

```
#include <iostream>

using namespace std;

int main() {
   int wallets;
   // Populate the wallets
   for(int i = 0; i < 20; ++i) {
      wallets[i] = 20 + i*10;
   }
   // Insertion sort implementation
   for(int i = 1; i < 20; ++i) {
      int key = wallets[i];
```

```
        int j = i - 1;
        // While sorting, move elements greater than key
        while(j >= 0 && wallets[j] > key) {
            wallets[j + 1] = wallets[j];
            j = j - 1;
        }
        wallets[j + 1] = key;
    }
    // Output sorted array
    cout << "Sorted wallets: ";
    for(int i = 0; i < 20; ++i) {
        cout << wallets[i] << " ";
    }
    cout << endl;
    return 0;
}
```

**Q2] 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.**

For sorting a small collection of integers with a known and limited range (money values: 0 to 6), counting sort is the ideal algorithm. Counting sort runs in linear time O(n)$O(n)$ when the range of values is small and is more efficient than comparison-based sorts for such cases. It is also simple and stable, preserving the order of equal elements.

**Why Counting Sort?**

- The number of wallets is small (n=10$n$=10).
- The value range is small (0 to 6).

- Counting sort avoids comparisons and sorts in O(n+k)$O(n+k)$, where k$k$ is the range (here k=7$k=7$).
- It is faster than quick sort, merge sort, or even insertion sort in this specific scenario.

```
#include <iostream>
using namespace std;
int main() {
    int wallets[11] = {2, 3, 0, 2, 1, 0, 2, 3, 4, 6};
    int count[7] = {0};  // To count occurrences from 0 to 6
    // Count the frequency of each wallet value
    for (int i = 0; i < 10; ++i) {
        count[wallets[i]]++;
    }
    // Write sorted values back to the array
    int index = 0;
    for (int i = 0; i < 7; ++i) {
        while (count[i] > 0) {
            wallets[index++] = i;
            count[i]--;
        }
    }
    // Print the sorted wallets
    cout << "Sorted wallets: ";
    for (int i = 0; i < 10; ++i) {
        cout << wallets[i] << " ";
    }
    cout << endl;
    return 0;
```

}

**Q3] 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.**

To arrange the scores in ascending order, the Quick Sort algorithm is indeed an efficient choice for an unsorted list with values spread across a wide range. Below is a C++ program to implement Quick Sort for your scenario.

```
#include <iostream>

using namespace std;

// Swap function

void swap(int* a, int* b) {

    int t = *a;

    *a = *b;

    *b = t;

}

// Partition function

int partition(int arr[], int low, int high) {

    int pivot = arr[high]; // choosing the last element as pivot

    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);
}

// QuickSort function
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);
    }
}
// Utility function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main() {
    int scores[11] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76};
    int n = 12;
    quickSort(scores, 0, n - 1);
    cout << "Sorted scores in ascending order: ";
    printArray(scores, n);
    return 0;
}
```

**Q4] 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.**

- QuickSort is chosen because it applies the divide-and-conquer principle but splits the array into *unequal* parts based on a pivot element.
- The array is recursively partitioned until all elements are sorted.
- Time complexity:
  - Best/Average case: O(nlogn)$O(n\log n)$
  - Worst case: O(n2)$O(n2)$ (if pivot choice is poor, e.g., already sorted input with last-element pivot).

```
#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 (Lomuto partition scheme)
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the 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);
}

// QuickSort function (Divide and Conquer)
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partition index
        quickSort(arr, low, pi - 1);        // Left subarray
        quickSort(arr, pi + 1, high);       // Right subarray
    }
}

// Display function
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

```
    cout << "Original deadlines:\n";
    printArray(deadlines, n);

    quickSort(deadlines, 0, n - 1);

    cout << "Sorted deadlines (ascending):\n";
    printArray(deadlines, n);

    return 0;
}
```

**Q5] 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.**

 When midpoints of a square are joined, the new square formed has **half the area** of the previous square.

 Thus, the sequence of areas becomes:

A, A2, A4, A8,…,A249A, \; \frac{A}{2}, \; \frac{A}{4}, \; \frac{A}{8}, \dots, \frac{A}{2^{49}}A,2A,4A,8A,…,249A

 The areas are stored in **sorted decreasing order**.

 Since the array is sorted, the **best searching approach is Binary Search (O(log n))**.

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

// Function for binary search in a descending sorted array
int binarySearch(double arr[], int n, double key) {
```

```
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid;  // Key found at index mid
        }
        else if (arr[mid] > key) {
            low = mid + 1;   // Search right half
        }
        else {
            high = mid - 1;  // Search left half
        }
    }
    return -1;  // Key not found
}

int main() {
    const int n = 50;
    double areas[n];

    // Initial area of SQ-1 (assuming side length = 1 unit)
    double initialArea = 1.0;
    areas[0] = initialArea;

    // Compute areas for SQ-2 to SQ-50 (each area is half of previous)
```

```
    for (int i = 1; i < n; i++) {
        areas[i] = areas[i - 1] / 2.0;
    }

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

    int result = binarySearch(areas, n, key);

    if (result != -1) {
        cout << "Area found at index " << result << " corresponding to SQ-" << (result + 1) << endl;
    } else {
        cout << "Area not found in the list of squares." << endl;
    }

    return 0;
}
```

**Q6] 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.**

To implement the chief guest meeting each player one by one using a linked list in C++, we can represent each player as a node in a singly linked list. The head coach introduces the first player (head node), then each player introduces the next player (node's next pointer), and so forth

```
#include <iostream>
```

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

// Node class for each player
class Player {
public:
    string name;
    Player* next;

    Player(string playerName) {
        name = playerName;
        next = nullptr;
    }
};

// Function to add a player to the end of the linked list
void addPlayer(Player*& head, string playerName) {
    Player* newPlayer = new Player(playerName);
    if (head == nullptr) {
        head = newPlayer;
        return;
    }
    Player* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newPlayer;
```

```
}

// Function to simulate the introductions and visiting sequence
void introducePlayers(Player* head) {
    Player* current = head;
    int playerNumber = 1;
    cout << "Chief guest meeting players:\n";
    while (current != nullptr) {
        if (playerNumber == 1)
            cout << "Head coach introduces Player " << playerNumber << ": " << current->name << "\n";
        else
            cout << "Player " << (playerNumber - 1) << " introduces Player " << playerNumber << ": " << current->name << "\n";

        cout << "Chief guest meets Player " << playerNumber << ": " << current->name << "\n";
        current = current->next;
        playerNumber++;
    }
}

int main() {
    Player* head = nullptr;

    int numPlayers;
    cout << "Enter number of players: ";
    cin >> numPlayers;
```

```
    cin.ignore();  // To consume newline after number input

    // Input player names and add to list
    for (int i = 1; i <= numPlayers; i++) {
        string playerName;
        cout << "Enter name of player " << i << ": ";
        getline(cin, playerName);
        addPlayer(head, playerName);
    }

    // Simulate the introduction and meeting sequence
    introducePlayers(head);

    // Free allocated memory (optional here, but good practice)
    Player* current = head;
    while (current != nullptr) {
        Player* next = current->next;
        delete current;
        current = next;
    }

    return 0;
}
```

**Q7] 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>
#include <string>
using namespace std;

class Node {
public:
    string stopName;
    Node* prev;
    Node* next;

    Node(string name) {
        stopName = name;
        prev = nullptr;
        next = nullptr;
    }
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
```

```
}

void append(string stopName) {
    Node* newNode = new Node(stopName);
    if (head == nullptr) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
}

void traverseForward() {
    cout << "Onward journey (A to D): ";
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->stopName;
        if (temp->next != nullptr) cout << " -> ";
        temp = temp->next;
    }
    cout << endl;
}

void traverseBackward() {
    cout << "Return journey (D to A): ";
    Node* temp = tail;
```

```
        while (temp != nullptr) {
            cout << temp->stopName;
            if (temp->prev != nullptr) cout << " -> ";
            temp = temp->prev;
        }
        cout << endl;
    }

    ~DoublyLinkedList() {
        Node* current = head;
        while (current != nullptr) {
            Node* next = current->next;
            delete current;
            current = next;
        }
    }
};

int main() {
    DoublyLinkedList busRoute;

    // Add bus stops
    busRoute.append("A");
    busRoute.append("B");
    busRoute.append("C");
    busRoute.append("D");
```

```
    // Show onward journey
    busRoute.traverseForward();

    // Show return journey
    busRoute.traverseBackward();

    return 0;
}
```

**Q8] 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, each with 2 Gullaks
    const int DaltaRows = 4;
    const int DaltaCols = 2;

    // Malta Gang: 2 members, each with 3 Gullaks
    const int MaltaRows = 2;
    const int MaltaCols = 3;

    // Sample data for Dalta Gang money
    int daltaGang[DaltaRows][DaltaCols] = {
```

```
        {10, 20},
        {15, 25},
        {30, 5},
        {45, 35}
    };

    // Sample data for Malta Gang money
    int maltaGang[MaltaRows][MaltaCols] = {
        {5, 10, 15},
        {20, 25, 30}
    };

    // Display Dalta Gang matrix
    cout << "Dalta Gang Money Matrix (" << DaltaRows << "x" << DaltaCols << "):\n";
    for (int i = 0; i < DaltaRows; i++) {
        for (int j = 0; j < DaltaCols; j++) {
            cout << daltaGang[i][j] << "\t";
        }
        cout << endl;
    }

    cout << "\nMalta Gang Money Matrix (" << MaltaRows << "x" << MaltaCols << "):\n";
    // Display Malta Gang matrix
    for (int i = 0; i < MaltaRows; i++) {
        for (int j = 0; j < MaltaCols; j++) {
            cout << maltaGang[i][j] << "\t";
```

```
        }
        cout << endl;
    }

    // Result matrix dimensions will be DaltaRows x MaltaCols
    int result[DaltaRows][MaltaCols] = {0};

    // Matrix multiplication
    // DaltaCols must be equal to MaltaRows for valid multiplication
    if (DaltaCols != MaltaRows) {
        cout << "\nMatrix multiplication not possible due to dimension
mismatch.\n";
        return 1;
    }

    for (int i = 0; i < DaltaRows; i++) {
        for (int j = 0; j < MaltaCols; j++) {
            result[i][j] = 0;
            for (int k = 0; k < DaltaCols; k++) {
                result[i][j] += daltaGang[i][k] * maltaGang[k][j];
            }
        }
    }

    // Display result matrix
    cout << "\nResult of Dalta Gang matrix multiplied by Malta Gang matrix ("
         << DaltaRows << "x" << MaltaCols << "):\n";
```

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

    return 0;
}
```

**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.**

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

struct Node {
    string name;
    Node* left;
    Node* right;

    Node(string val) : name(val), left(nullptr), right(nullptr) {}
};
```

```
// Insert node into BST
Node* insert(Node* root, string name) {
    if (!root)
        return new Node(name);
    if (name < root->name)
        root->left = insert(root->left, name);
    else if (name > root->name)
        root->right = insert(root->right, name);
    return root;
}

// Inorder traversal to display BST
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->name << " ";
        inorder(root->right);
    }
}

// Find minimum node in BST (leftmost)
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
```

```
}

// Find node in BST by name
Node* findNode(Node* root, string name) {
    if (!root || root->name == name)
        return root;
    if (name < root->name)
        return findNode(root->left, name);
    else
        return findNode(root->right, name);
}

// Find successor of given node (inorder successor)
Node* findSuccessor(Node* root, Node* node) {
    if (node->right)
        return minValueNode(node->right);

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

```
}

int main() {
    Node* root = nullptr;

    // Insert given family member names into BST
    string names[] = {"Q", "S", "R", "T", "M", "A", "B", "P", "N"};
    for (string name : names) {
        root = insert(root, name);
    }

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

    // Find node with name starting M
    Node* target = findNode(root, "M");
    if (!target) {
        cout << "Member starting with M not found\n";
        return 0;
    }

    // Find and display successor
    Node* successor = findSuccessor(root, target);
    if (successor)
        cout << "Successor of " << target->name << " is " << successor->name <<
endl;
    else
```

```
        cout << target->name << " has no successor in BST\n";

    return 0;
}
```

**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;

// Structure for BST node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a new node in BST
Node* insert(Node* root, int val) {
    if (root == nullptr) {
        return new Node(val);
    }
```

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

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

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

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

```
    postorder(root->right);
    cout << root->data << " ";
}

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

    // Insert values into BST
    for (int val : values) {
        root = insert(root, val);
    }

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

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

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

    return 0;
}
```

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

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

// Structure for BST node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};

// Insert a node in BST
Node* insert(Node* root, int val) {
    if (root == nullptr)
        return new Node(val);
    if (val < root->data)
        root->left = insert(root->left, val);
    else if (val > root->data)
        root->right = insert(root->right, val);
    return root;
}
```

```
// Search an element in BST
Node* search(Node* root, int key) {
    if (root == nullptr || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

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

    // Insert values into the BST
    for (int val : values) {
        root = insert(root, val);
    }

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

    Node* found = search(root, key);

    if (found)
```

```
        cout << "Element " << key << " found in the BST.\n";
    else
        cout << "Element " << key << " not found in the BST.\n";


    return 0;
}
```

**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.**

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


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


    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};
```

```
// Function to insert a value in BST
Node* insertNode(Node* root, int val) {
    if (root == nullptr) {
        return new Node(val);
    }

    if (val < root->data) {
        root->left = insertNode(root->left, val);
    } else if (val > root->data) {
        root->right = insertNode(root->right, val);
    }

    return root;
}

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

int main() {
    int rollNumbers[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};
```

```
    int n = sizeof(rollNumbers) / sizeof(rollNumbers[0]);

    Node* root = nullptr;

    // Insert roll numbers into BST
    for (int i = 0; i < n; i++) {
        root = insertNode(root, rollNumbers[i]);
    }

    // Display roll numbers in ascending order using in-order traversal
    cout << "Roll numbers in ascending order:\n";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
#include <iostream>
using namespace std;

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

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};
```

```
// Function to insert nodes in BST
Node* insertNode(Node* root, int val) {
    if (root == nullptr)
        return new Node(val);
    if (val < root->data)
        root->left = insertNode(root->left, val);
    else if (val > root->data)
        root->right = insertNode(root->right, val);
    return root;
}

// In-order traversal (ascending order)
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

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

    Node* root = nullptr;
```

```
    // Construct BST from roll numbers
    for (int i = 0; i < n; i++) {
        root = insertNode(root, rollNumbers[i]);
    }

    cout << "Roll numbers in ascending order:" << endl;
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

**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.**

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
```

```
    }
};

// Function to insert nodes in BST
Node* insertNode(Node* root, int val) {
    if (root == nullptr)
        return new Node(val);
    if (val < root->data)
        root->left = insertNode(root->left, val);
    else if (val > root->data)
        root->right = insertNode(root->right, val);
    return root;
}

// Find minimum node in BST (leftmost node)
Node* findMin(Node* root) {
    while (root && root->left != nullptr)
        root = root->left;
    return root;
}

// Function to delete a node from BST
Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    if (key < root->data)
```

```
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children: Get inorder successor (smallest in right subtree)
        Node* temp = findMin(root->right);
        root->data = temp->data;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Inorder traversal to display BST
void inorder(Node* root) {
```

```
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    int n = sizeof(values) / sizeof(values[0]);

    // Insert values into BST
    for (int i = 0; i < n; i++) {
        root = insertNode(root, values[i]);
    }

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

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

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

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

    return 0;
}
```

**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.**

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

struct Node {
    string name;
    Node* left;
    Node* right;

    Node(string val) {
        name = val;
        left = nullptr;
        right = nullptr;
    }
};

// Insert family member based on lexicographical order
Node* insert(Node* root, string name) {
    if (root == nullptr)
        return new Node(name);
```

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

int main() {
    Node* root = nullptr;

    // Insert family members into BST
    string members[] = {"John", "Alice", "Robert", "Zara", "Mary", "David",
"Emily"};
    for (const string& member : members) {
        root = insert(root, member);
    }

    cout << "Family tree hierarchy (In-order traversal): ";
```

```
    inorder(root);
    cout << "\n";

    cout << "Family tree hierarchy (Pre-order traversal): ";
    preorder(root);
    cout << "\n";

    cout << "Family tree hierarchy (Post-order traversal): ";
    postorder(root);
    cout << "\n";

    // Search for a family member
    string searchName;
    cout << "Enter a family member name to search: ";
    getline(cin, searchName);

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

    return 0;
}
```