Q7

Question:

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.

Answer (best technique & code):

Because the data is already sorted (or nearly sorted), **Insertion Sort** is ideal — it is simple and efficient on already sorted or nearly-sorted data (O(n) for already sorted). Below is a C++ program implementing insertion sort with line-by-line comments.

```
// Q7: Insertions Sort for nearly-sorted data (20 wallets increasing
by 10)
// Insertion Sort is preferred here because the array is already
sorted -> O(n) best-case.

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

int main() {
    // create the wallet array: 20,30,40,... for 20 friends
    vector<int> arr;
    for (int i = 0; i < 20; ++i) {
        arr.push_back(20 + i * 10);    // 20 + 0*10, 20 + 1*10, ...
    }
    // Print original (already sorted)
    cout << "Original wallets: ";
    for (int v : arr) cout << v << " ";
    cout << "\n";

    // Insertion sort implementation
    int n = arr.size();                // number of elements
    for (int i = 1; i < n; ++i) {      // start from second element
        int key = arr[i];              // element to be inserted into
```

```
sorted left side
        int j = i - 1;                  // start comparing from the
element before key
        // shift elements that are greater than key to right
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];        // move element right
            --j;                        // move left
        }
        arr[j + 1] = key;               // insert key at correct
position
    }

    // Print sorted array (should be same as original)
    cout << "Sorted wallets (Insertion Sort): ";
    for (int v : arr) cout << v << " ";
    cout << "\n";
    return 0;
}
```

⏱️ **Time Complexity:** O(n²) average/worst, O(n) best (already sorted)
💾 **Space Complexity:** O(1)

Q8

Question:
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.

Answer (best technique & code):
Because keys are small-range integers (0..6), **Counting Sort** (a non-comparative linear-time algorithm) is ideal. It runs in O(n + k), with k small.

```
// Q8: Counting Sort for small range data (0..6)

#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
    // Given data: 3 wallets -> $2, 2 wallets -> $3, 2 wallets -> $0,
1 -> $1, 1 -> $4
    vector<int> wallets = {2,2,2,3,3,0,0,1,4,2}; // sample order (10
elements)
    // Determine range (0..6)
    int k = 7;                          // possible values 0..6
inclusive
    vector<int> count(k, 0);            // frequency array initialized
to 0

    // Count occurrences
    for (int v : wallets) {
        ++count[v];                     // increment count for value v
    }

    // Build sorted array using counts
    vector<int> sorted;
    for (int val = 0; val < k; ++val) {
        while (count[val]-- > 0) {
            sorted.push_back(val);      // append value val count[val]
times
        }
    }

    // Print result
    cout << "Original wallets: ";
    for (int v : wallets) cout << v << " ";
    cout << "\nSorted wallets (Counting Sort): ";
    for (int v : sorted) cout << v << " ";
    cout << "\n";
    return 0;
}
```

⏱ **Time Complexity:** O(n + k) 💾 **Space Complexity:** O(k)

Q9

Question:

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.

Answer (code):

Implement the standard QuickSort (last element pivot) with comments.

```
// Q9: Quick Sort implementation for an unsorted score list

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

// Partition using last element as pivot, returns pivot index
int partition(vector<int>& a, int l, int r) {
    int pivot = a[r];                 // choose last element as pivot
    int i = l - 1;                    // index of smaller element
    for (int j = l; j <= r - 1; ++j) {
        if (a[j] <= pivot) {          // if current element <= pivot
            ++i;
            swap(a[i], a[j]);         // place it to left partition
        }
    }
    swap(a[i + 1], a[r]);             // put pivot after smaller
elements
    return i + 1;                     // return pivot index
}

// QuickSort recursive
void quickSort(vector<int>& a, int l, int r) {
    if (l < r) {
        int p = partition(a, l, r);   // partition index
        quickSort(a, l, p - 1);       // sort left subarray
```

```
        quickSort(a, p + 1, r);       // sort right subarray
    }
}

int main() {
    vector<int> scores = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32,
76};
    cout << "Original scores: ";
    for (int s : scores) cout << s << " ";
    cout << "\n";

    quickSort(scores, 0, (int)scores.size() - 1); // perform QuickSort

    cout << "Sorted scores (ascending): ";
    for (int s : scores) cout << s << " ";
    cout << "\n";
    return 0;
}
```

⏱ **Time Complexity:** O(n log n) avg | O(n^2) worst 💾 **Space:** O(log n)

Q10

Question:

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.

Answer (interpretation & code):

A divide-and-conquer technique that divides into unequal parts describes **Quick Sort**. We'll implement QuickSort (same as above).

```
// Q10: Quick Sort for project deadlines (unequal partitioning typical
for QuickSort)

#include <iostream>
```

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

int partition(vector<int>& a, int l, int r) {
    int pivot = a[r];                 // pivot as last element
    int i = l - 1;
    for (int j = l; j <= r - 1; ++j) {
        if (a[j] <= pivot) {
            ++i;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[r]);
    return i + 1;
}

void quickSort(vector<int>& a, int l, int r) {
    if (l < r) {
        int p = partition(a, l, r);
        quickSort(a, l, p - 1);
        quickSort(a, p + 1, r);
    }
}

int main() {
    vector<int> deadlines = {25, 12, 45, 7, 30, 18, 40, 22, 10, 35};
    cout << "Original deadlines: ";
    for (int d : deadlines) cout << d << " ";
    cout << "\n";

    quickSort(deadlines, 0, (int)deadlines.size() - 1);

    cout << "Sorted deadlines (ascending): ";
    for (int d : deadlines) cout << d << " ";
    cout << "\n";
    return 0;
}
```

⏱️ **Time:** O(n log n) 💾 **Space:** O(log n)

Q11

Question:

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.

Answer (best method & code):

The areas formed by repeatedly connecting midpoints form a strictly decreasing geometric sequence (each new square area is 1/2 of previous if side halves? — actually connecting midpoints of a square produces a square with area = 1/2 of original). If the array is sorted (it will be sorted by construction if we store them in sequence), the best search is **Binary Search** (O(log n)). We'll assume the areas array is sorted decreasing or increasing; we can store ascending and run binary search.

```
// Q11: Binary Search on array of 50 square areas
// We assume SQ1 area = A0 (given or assume 1), SQi area = A0 *
(1/2)^(i-1)
// We'll build array ascending and binary search for target.

#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

int binarySearch(const vector<double>& a, double target) {
    int l = 0, r = (int)a.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (fabs(a[mid] - target) < 1e-9) return mid; // found
(floating tolerance)
        if (a[mid] < target) l = mid + 1;             // move right if
mid < target
        else r = mid - 1;
    }
    return -1; // not found
}
```

```
int main() {
    double sq1_area = 1024.0;            // example area for SQ-1 (can
be any positive)
    int n = 50;
    vector<double> areas;
    // Create areas: each next square has half the area of previous
(area ratio = 1/2)
    for (int i = 0; i < n; ++i) {
        double area = sq1_area * pow(0.5, i); // area of SQ-(i+1)
        areas.push_back(area);
    }

    // Sort ascending to apply standard binary search
    sort(areas.begin(), areas.end());

    // Example input: read target area from user
    double target;
    cout << "Enter area to search: ";
    cin >> target;

    int idx = binarySearch(areas, target);
    if (idx == -1) cout << "Area not found\n";
    else cout << "Area found at index " << idx << " (0-based) with
value " << areas[idx] << "\n";
    return 0;
}
```

⏱️ **Time Complexity:** O(log n) 💾 **Space Complexity:** O(1)

Q12

Question:

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.

Answer (singly linked list simulation & code):

This is an obvious representation by a singly linked list where each node is a player and traversal simulates introductions. We will build the list and then traverse.

```
// Q12: Singly linked list to model sequential introductions

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

struct Player {
    string name;       // player name
    Player* next;      // pointer to next player
    Player(const string& s) : name(s), next(nullptr) {}
};

int main() {
    // Build the list of players (example)
    Player* head = new Player("Player1");
    head->next = new Player("Player2");
    head->next->next = new Player("Player3");
    head->next->next->next = new Player("Player4");
    head->next->next->next->next = new Player("Player5");

    // Simulate introductions: head coach introduces head, then
traversal
    cout << "Head Coach: Introducing " << head->name << "\n";
    Player* cur = head->next;           // player who will next
introduce others
    Player* guest = head;               // chief guest meets players
in sequence
    // Chief guest meets head already, now move forward
    while (cur != nullptr) {
        cout << cur->name << " is introduced by previous player.\n";
        cout << "Chief Guest meets " << cur->name << "\n";
        cur = cur->next;               // move to next introduction
```

```
    }

    // Free memory (good practice)
    while (head) {
        Player* tmp = head;
        head = head->next;
        delete tmp;
    }
    return 0;
}
```

⏱️ **Time:** O(n) 💾 **Space:** O(n) (for list nodes)

Q5

Question:

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.

Answer (code):

Use a doubly linked list; forward traversal prints onward, backward prints return.

```
// Q5: Doubly linked list to model bus journey A->B->C->D and back D->C->B->A

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

struct Stop {
    string name;     // stop name
    Stop* prev;      // previous pointer
    Stop* next;      // next pointer
    Stop(const string& s) : name(s), prev(nullptr), next(nullptr) {}
};
```

```
int main() {
    // Create stops
    Stop* A = new Stop("A");
    Stop* B = new Stop("B");
    Stop* C = new Stop("C");
    Stop* D = new Stop("D");

    // Link them: A <-> B <-> C <-> D
    A->next = B; B->prev = A;
    B->next = C; C->prev = B;
    C->next = D; D->prev = C;

    // Forward traversal (onward)
    cout << "Onward journey: ";
    Stop* cur = A;
    while (cur != nullptr) {
        cout << cur->name;
        if (cur->next) cout << " -> ";
        cur = cur->next;
    }
    cout << "\n";

    // Backward traversal (return) - start from D
    cout << "Return journey: ";
    cur = D;
    while (cur != nullptr) {
        cout << cur->name;
        if (cur->prev) cout << " -> ";
        cur = cur->prev;
    }
    cout << "\n";

    // Free memory
    delete A; delete B; delete C; delete D;
    return 0;
}
```

⏱️ **Time:** O(n) traversal 💾 **Space:** O(n)

Q6

Question:
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.
- Multiply Dalta Gang matrix with Malta Gang Matrix.

Answer (code & explanation):
This models matrix multiplication; check dimensions carefully: Dalta is 4×2, Malta is 2×3 -> result 4×3.

```
// Q6: Matrix display and multiplication (Dalta: 4x2, Malta: 2x3 ->
result 4x3)

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

int main() {
    // Dalta Gang: 4 members x 2 gullaks each -> 4x2 matrix
    vector<vector<int>> Dalta = {
        {10, 20},   // member1's two gullaks
        {5, 15},    // member2
        {7, 8},     // member3
        {12, 4}     // member4
    };

    // Malta Gang: 2 members x 3 gullaks each -> 2x3 matrix
    vector<vector<int>> Malta = {
        {2, 3, 4},  // member1's three gullaks
        {1, 5, 6}   // member2
    };

    // Display Dalta matrix
    cout << "Dalta Gang (4x2):\n";
```

```
    for (auto &row : Dalta) {
        for (int val : row) cout << val << "\t";
        cout << "\n";
    }

    // Display Malta matrix
    cout << "\nMalta Gang (2x3):\n";
    for (auto &row : Malta) {
        for (int val : row) cout << val << "\t";
        cout << "\n";
    }

    // Multiply Dalta (4x2) * Malta (2x3) -> result 4x3
    int r1 = 4, c1 = 2, r2 = 2, c2 = 3;
    vector<vector<int>> res(r1, vector<int>(c2, 0));

    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            for (int k = 0; k < c1; ++k) {
                res[i][j] += Dalta[i][k] * Malta[k][j]; // sum of
products
            }
        }
    }

    // Display result
    cout << "\nDalta x Malta (4x3):\n";
    for (auto &row : res) {
        for (int val : row) cout << val << "\t";
        cout << "\n";
    }

    return 0;
}
```

⏱ **Time:** $O(r_1 \times c_1 \times c_2) = O(n^3)$ 💾 **Space:** $O(r_1 \times c_2)$

Q13

Question:

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.

Answer (code):

We'll use single-letter names as strings and standard BST lexicographic ordering. To find successor (inorder successor) of node with key "M", find the node and compute its successor.

```
// Q13: BST for names (single letters), insert <Q,S,R,T,M,A,B,P,N> and
find successor of "M"

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

struct Node {
    string key;
    Node* left;
    Node* right;
    Node(const string& k) : key(k), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, const string& key) {
    if (!root) return new Node(key);            // create new node
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    return root;
}

// find node with given key
Node* findNode(Node* root, const string& key) {
    if (!root) return nullptr;
```

```
    if (root->key == key) return root;
    if (key < root->key) return findNode(root->left, key);
    return findNode(root->right, key);
}

// find minimum node in subtree
Node* findMin(Node* root) {
    while (root && root->left) root = root->left;
    return root;
}

// find inorder successor of given key in BST
Node* inorderSuccessor(Node* root, const string& key) {
    Node* current = findNode(root, key);
    if (!current) return nullptr;

    // If right subtree exists, successor is min in right subtree
    if (current->right) return findMin(current->right);

    // Otherwise, successor is lowest ancestor for which current is in
left subtree
    Node* succ = nullptr;
    Node* ancestor = root;
    while (ancestor != current) {
        if (current->key < ancestor->key) {
            succ = ancestor;              // this ancestor might be
successor
            ancestor = ancestor->left;
        } else ancestor = ancestor->right;
    }
    return succ;
}

int main() {
    string keys[] = {"Q","S","R","T","M","A","B","P","N"};
    Node* root = nullptr;
    for (auto &k : keys) root = insert(root, k);

    string target = "M";
```

```
    Node* succ = inorderSuccessor(root, target);
    if (succ) cout << "Successor of " << target << " is " << succ->key
<< "\n";
    else cout << "No successor found for " << target << "\n";
    return 0;
}
```

⏱️ **Time:** O(h) (h = tree height) 💾 **Space:** O(h)

Q17

Question:

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

Answer (code):

Standard recursive traversals with comments.

```
// Q17: Implement inorder, preorder, postorder traversals for a BST

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}
```

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

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

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

int main() {
    // Example BST
    int keys[] = {50, 30, 20, 40, 70, 60, 80};
    Node* root = nullptr;
    for (int k : keys) root = insert(root, k);

    cout << "Inorder: "; inorder(root); cout << "\n";
    cout << "Preorder: "; preorder(root); cout << "\n";
    cout << "Postorder: "; postorder(root); cout << "\n";
    return 0;
}
```

⏱️ **Time:** O(n) 💾 **Space:** O(h)

Q3

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

Answer (code):
Implement BST search (iterative and recursive possible). We'll show iterative with comments.

```
// Q3: Search element in BST (iterative)

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// Iterative search function
bool searchBST(Node* root, int key) {
    Node* cur = root;
    while (cur) {
        if (cur->data == key) return true;      // found
        if (key < cur->data) cur = cur->left;   // go left
        else cur = cur->right;                  // go right
    }
    return false;                               // not found
}
```

```
int main() {
    int keys[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};
    Node* root = nullptr;
    for (int k : keys) root = insert(root, k);

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

    if (searchBST(root, target)) cout << target << " found in BST\n";
    else cout << target << " not found\n";
    return 0;
}
```

⏱ **Time:** O(h) 💾 **Space:** O(1)

Q4

Question:

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.

Answer (code):

Construct BST and perform inorder.

```
// Q4: Build BST from given roll numbers and display inorder
(ascending)

#include <iostream>
using namespace std;

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

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

Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {
    int rolls[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};
    Node* root = nullptr;
    for (int r : rolls) root = insert(root, r);

    cout << "Students roll numbers in ascending order (Inorder): ";
    inorder(root); cout << "\n";
    return 0;
}
```

⏱ **Time:** O(n) 💾 **Space:** O(h)

Q19

Question:

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.

Answer (code):

BST deletion with three cases handled.

```
// Q19: BST deletion (user input key)

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int v) : data(v), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

Node* findMin(Node* root) {
    while (root && root->left) root = root->left;
    return root;
}

Node* deleteNode(Node* root, int key) {
    if (!root) return nullptr;
    if (key < root->data) root->left = deleteNode(root->left, key);
    else if (key > root->data) root->right = deleteNode(root->right,
key);
    else {
        // Node found
        if (!root->left && !root->right) {
            delete root; return nullptr;            // leaf
        } else if (!root->left) {
            Node* tmp = root->right; delete root; return tmp; // one
right child
        } else if (!root->right) {
            Node* tmp = root->left; delete root; return tmp; // one
left child
```

```
        } else {
            Node* succ = findMin(root->right);    // two children: use
inorder successor
            root->data = succ->data;
            root->right = deleteNode(root->right, succ->data);
        }
    }
    return root;
}

void inorder(Node* root) { if (!root) return; inorder(root->left);
cout<<root->data<<" "; inorder(root->right); }

int main() {
    int rolls[] = {50,30,70,20,40,60,80};
    Node* root = nullptr;
    for (int r : rolls) root = insert(root, r);

    cout << "Original inorder: "; inorder(root); cout << "\n";

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

    root = deleteNode(root, key);

    cout << "Inorder after deletion: "; inorder(root); cout << "\n";
    return 0;
}
```

⏱️ **Time:** O(h) 💾 **Space:** O(h)

Q23

Question:

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:

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

Answer (code):

BST keyed by name; includes insert, search, and traversals.

```
// Q23: Family tree using BST (keyed by name). Insert, traverse
(in/pre/post), search.

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

struct Node {
    string name;
    Node* left;
    Node* right;
    Node(const string& s) : name(s), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, const 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);
    // duplicates ignored for simplicity
    return root;
}

bool search(Node* root, const string& name) {
    if (!root) return false;
    if (root->name == name) return true;
    if (name < root->name) return search(root->left, name);
    return search(root->right, name);
}
```

```
void inorder(Node* r) { if(!r) return; inorder(r->left); cout<<r-
>name<<" "; inorder(r->right); }
void preorder(Node* r) { if(!r) return; cout<<r->name<<" ";
preorder(r->left); preorder(r->right); }
void postorder(Node* r) { if(!r) return; postorder(r->left);
postorder(r->right); cout<<r->name<<" "; }

int main() {
    string family[] =
{"John","Alice","Bob","Eve","Charlie","David","Grace"};
    Node* root = nullptr;
    for (auto &n : family) root = insert(root, n);

    cout << "Inorder (alphabetical): "; inorder(root); cout << "\n";
    cout << "Preorder: "; preorder(root); cout << "\n";
    cout << "Postorder: "; postorder(root); cout << "\n";

    string query;
    cout << "Enter name to search: ";
    cin >> query;
    if (search(root, query)) cout << query << " found in family
tree\n";
    else cout << query << " not found\n";
    return 0;
}
```

⏱ **Time:** O(h) per operation 💾 **Space:** O(h)

Q. Apply the adjecency linked list method to store the given graph into memory

C++ Code: Adjacency Linked List

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

// Node structure for linked list
struct Node {
    int vertex;
    Node* next;
};
```

```
// Graph structure
struct Graph {
    int V;        // Number of vertices
    Node** head;  // Array of linked list heads
};

// Function to create a new node
Node* createNode(int v) {
    Node* newNode = new Node;
    newNode->vertex = v;
    newNode->next = nullptr;
    return newNode;
}

// Function to create a graph with V vertices
Graph* createGraph(int V) {
    Graph* graph = new Graph;
    graph->V = V;
    graph->head = new Node*[V];
    for (int i = 0; i < V; i++)
        graph->head[i] = nullptr;
    return graph;
}

// Function to add an edge (undirected)
void addEdge(Graph* graph, int src, int dest) {
    // Add edge from src to dest
    Node* newNode = createNode(dest);
    newNode->next = graph->head[src];
    graph->head[src] = newNode;

    // Since undirected, add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->head[dest];
    graph->head[dest] = newNode;
}

// Function to print the adjacency list
void printGraph(Graph* graph) {
```

```
    for (int v = 0; v < graph->V; v++) {
        Node* temp = graph->head[v];
        cout << "Vertex " << v << ":";
        while (temp) {
            cout << " -> " << temp->vertex;
            temp = temp->next;
        }
        cout << endl;
    }
}

// Driver code
int main() {
    int V = 4; // Number of vertices
    Graph* graph = createGraph(V);

    // Add edges
    addEdge(graph, 0, 1); // A-B
    addEdge(graph, 0, 2); // A-C
    addEdge(graph, 1, 2); // B-C
    addEdge(graph, 2, 3); // C-D

    // Print adjacency list
    printGraph(graph);

    return 0;
}
```

Explanation

- Vertices are numbered `0, 1, 2, 3` (you can map 0→A, 1→B, etc.).
- Each vertex has a linked list of its neighbors.
- Adding an edge involves inserting at the **head of the linked list** for efficiency.
- `printGraph` shows the adjacency lists.

Q. Apply the adjecency matrix method to store given graph in the memory

Adjacency Matrix Representation

- An **adjacency matrix** is a 2D array V x V where V is the number of vertices.
- If there is an edge between vertex `i` and vertex `j`, `matrix[i][j] = 1` (or the weight if weighted).
- If no edge exists, `matrix[i][j] = 0`.
- Works well for **dense graphs**.

C++ Code for Adjacency Matrix

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

int main() {
    int V = 4; // Number of vertices

    // Create a V x V matrix and initialize with 0
    int matrix[V][V] = {0};

    // Add edges (undirected)
    matrix[0][1] = 1; // A-B
    matrix[1][0] = 1; // B-A

    matrix[0][2] = 1; // A-C
    matrix[2][0] = 1; // C-A

    matrix[1][2] = 1; // B-C
    matrix[2][1] = 1; // C-B

    matrix[2][3] = 1; // C-D
    matrix[3][2] = 1; // D-C

    // Print adjacency matrix
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
```

```
    }

    return 0;
}
```

Q. How we declare the Graph in C++ kindly write the code for the same.

Using Adjacency Matrix

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

int main() {
    int V = 4; // Number of vertices

    // Declare a 2D array (matrix) for the graph
    int graph[V][V] = {0}; // Initialize all elements to 0

    // Example: Add edges (undirected graph)
    graph[0][1] = 1; // A-B
    graph[1][0] = 1; // B-A

    graph[0][2] = 1; // A-C
    graph[2][0] = 1; // C-A

    graph[1][2] = 1; // B-C
    graph[2][1] = 1; // C-B

    graph[2][3] = 1; // C-D
    graph[3][2] = 1; // D-C

    // Print the graph
    for(int i = 0; i < V; i++) {
        for(int j = 0; j < V; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
```

```
    return 0;
}
```

✅ This creates a **V×V adjacency matrix** where `graph[i][j] = 1` means an edge exists.

Q. In-Order à [E, B, J, F, A, H, D, K] and Pre Order [A, B, E, F, J, D, H, K] Construct the Binary Tree

C++ Code to Construct Binary Tree

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

// Node structure
struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char val) : data(val), left(nullptr), right(nullptr) {}
};

// Helper function to build tree recursively
Node* buildTree(char preorder[], int preStart, int preEnd,
                char inorder[], int inStart, int inEnd,
                unordered_map<char,int>& inMap) {
    if(preStart > preEnd || inStart > inEnd)
        return nullptr;

    char rootVal = preorder[preStart];
    Node* root = new Node(rootVal);

    int inRoot = inMap[rootVal];
    int leftTreeSize = inRoot - inStart;

    root->left = buildTree(preorder, preStart+1,
preStart+leftTreeSize,
                           inorder, inStart, inRoot-1, inMap);
```

```
    root->right = buildTree(preorder, preStart+leftTreeSize+1, preEnd,
                            inorder, inRoot+1, inEnd, inMap);
    return root;
}

// Utility function to print Inorder (for verification)
void printInorder(Node* root) {
    if(!root) return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

int main() {
    char inorder[]  = {'E','B','J','F','A','H','D','K'};
    char preorder[] = {'A','B','E','F','J','D','H','K'};
    int n = sizeof(inorder)/sizeof(inorder[0]);

    unordered_map<char,int> inMap;
    for(int i=0;i<n;i++)
        inMap[inorder[i]] = i;

    Node* root = buildTree(preorder, 0, n-1, inorder, 0, n-1, inMap);

    cout << "Inorder of constructed tree: ";
    printInorder(root);
    cout << endl;

    return 0;
}
```

Q. Write a C++ Program to insert the given node in a Binary search tree, 20, 10, 28. 35, 30, 5, 12

C++ Program to Insert Nodes into a BST

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

```
// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

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

    if(key < root->data)
        root->left = insert(root->left, key);
    else if(key > root->data)
        root->right = insert(root->right, key);

    return root;
}

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

int main() {
    Node* root = nullptr;

    // Insert given nodes
    int nodes[] = {20, 10, 28, 35, 30, 5, 12};
    int n = sizeof(nodes)/sizeof(nodes[0]);

    for(int i = 0; i < n; i++)
        root = insert(root, nodes[i]);
```

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

    return 0;
}
```

Q. Write a C++ program to search the successor of a node in Binary serch tree.

C++ Program to Find Successor of a Node in BST

Successor of a node = the smallest node greater than the given node.

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

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

// Insert function (same as above)
Node* insert(Node* root, int key) {
    if(root == nullptr) return new Node(key);
    if(key < root->data) root->left = insert(root->left, key);
    else if(key > root->data) root->right = insert(root->right, key);
    return root;
}

// Find minimum value in a tree
Node* findMin(Node* root) {
    while(root && root->left)
        root = root->left;
    return root;
}

// Find successor of a given key
```

```
Node* successor(Node* root, int key) {
    Node* succ = nullptr;
    Node* current = root;

    while(current) {
        if(key < current->data) {
            succ = current;
            current = current->left;
        } else if(key > current->data) {
            current = current->right;
        } else {
            // Node found
            if(current->right)
                succ = findMin(current->right);
            break;
        }
    }
    return succ;
}

int main() {
    Node* root = nullptr;
    int nodes[] = {20, 10, 28, 35, 30, 5, 12};
    int n = sizeof(nodes)/sizeof(nodes[0]);

    for(int i = 0; i < n; i++)
        root = insert(root, nodes[i]);

    int key;
    cout << "Enter the node to find its successor: ";
    cin >> key;

    Node* succ = successor(root, key);
    if(succ)
        cout << "Successor of " << key << " is " << succ->data <<
endl;
    else
        cout << "No successor found (it is the largest node)." <<
endl;
```

```
    return 0;
}
```

Q. Write a C++ program to search the maximum element in a BST

In a BST, the **maximum element** is always the **rightmost node.**

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

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

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

// Find maximum
Node* findMax(Node* root) {
    if(!root) return nullptr;
    while(root->right)
        root = root->right;
    return root;
}

int main() {
    Node* root = nullptr;
    int nodes[] = {20, 10, 28, 35, 30, 5, 12};
    for(int key : nodes) root = insert(root, key);

    Node* maxNode = findMax(root);
    if(maxNode) cout << "Maximum element in BST: " << maxNode->data <<
```

```
endl;
    return 0;
}
```

Q. Write a C++ program to search the minimum element in a BST

Similarly, the **minimum element** is the **leftmost node.**

```
Node* findMin(Node* root) {
    if(!root) return nullptr;
    while(root->left)
        root = root->left;
    return root;
}
```

Q. Construct the Binary tree with help of the given traversal details.
In-order: [40, 20, 50, 10, 60, 30] Preorder: [10, 20, 40, 50, 30, 60]

Construct Binary Tree from Inorder and Preorder

Given:

- Inorder: [40, 20, 50, 10, 60, 30]
- Preorder: [10, 20, 40, 50, 30, 60]

Code:

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

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

Node* buildTree(int preorder[], int preStart, int preEnd,
                int inorder[], int inStart, int inEnd,
                unordered_map<int,int>& inMap) {
    if(preStart > preEnd || inStart > inEnd) return nullptr;
```

```
    int rootVal = preorder[preStart];
    Node* root = new Node(rootVal);
    int inRoot = inMap[rootVal];
    int leftSize = inRoot - inStart;
    root->left = buildTree(preorder, preStart+1, preStart+leftSize,
inorder, inStart, inRoot-1, inMap);
    root->right = buildTree(preorder, preStart+leftSize+1, preEnd,
inorder, inRoot+1, inEnd, inMap);
    return root;
}

void inorderPrint(Node* root) {
    if(!root) return;
    inorderPrint(root->left);
    cout << root->data << " ";
    inorderPrint(root->right);
}

int main() {
    int inorderArr[] = {40, 20, 50, 10, 60, 30};
    int preorderArr[] = {10, 20, 40, 50, 30, 60};
    int n = sizeof(inorderArr)/sizeof(inorderArr[0]);
    unordered_map<int,int> inMap;
    for(int i=0;i<n;i++) inMap[inorderArr[i]] = i;

    Node* root = buildTree(preorderArr, 0, n-1, inorderArr, 0, n-1,
inMap);
    cout << "Inorder of constructed tree: ";
    inorderPrint(root);
    cout << endl;
}
```

Q. Explain the In-order traversal with help of given examle, write a C++ program to implement the In-order traversal

C++ Program for In-order Traversal

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

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

// In-order traversal
void inorder(Node* root) {
    if(!root) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Sample BST insert
Node* insert(Node* root, int key) {
    if(!root) return new Node(key);
    if(key < root->data) root->left = insert(root->left, key);
    else if(key > root->data) root->right = insert(root->right, key);
    return root;
}

int main() {
    Node* root = nullptr;
    int nodes[] = {10,5,20,3,7,15,25};
    for(int key : nodes) root = insert(root,key);

    cout << "In-order traversal: ";
    inorder(root);
    cout << endl;
}
```

Q. Explain the Pre-order traversal with help of given examle, write a C++ program to implement the Pre-order traversal

C++ Program for Pre-order Traversal

```
// Pre-order traversal
void preorder(Node* root) {
```

```
    if(!root) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
```

Q. Explain the Post-order traversal with help of given examle, write a C++ program to implement the Post-order -order traversal

Definition:

- **Left → Right → Root**
- Visits left subtree, then right subtree, then the root.

Example Tree: (Same as above)

Post-order Traversal: 3, 7, 5, 15, 25, 20, 10

C++ Program for Post-order Traversal

```
// Post-order traversal
void postorder(Node* root) {
    if(!root) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}
```

Q. Explain the key propeties of a red-black tree, How we can declare the structure of a red-Black tree in C++?

Key Properties:

1. Every node is either **Red or Black**.
2. **Root node** is always **Black**.
3. **Red node cannot have a red child** (no two reds in a row).
4. Every path from a node to its leaf **contains same number of black nodes**.
5. Leaves (NULL) are considered **Black**.

Purpose: Ensures tree remains **approximately balanced**, guaranteeing O(log n) search, insert, delete.

C++ Structure Declaration

```
enum Color { RED, BLACK };

struct RBNode {
    int data;
    Color color;
    RBNode* left;
    RBNode* right;
    RBNode* parent;

    RBNode(int val) : data(val), color(RED), left(nullptr),
right(nullptr), parent(nullptr) {}
};
```

- color stores the node color.
- parent pointer is used for insertion and rotation.
- left and right are child pointers.

✅ This is the **basic structure** for Red-Black Tree nodes in C++.