1. **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.**

- Insertion Sort works best for nearly sorted or small data because it reduces unnecessary comparisons, making it very efficient in such cases.
- It is simple to implement and mimics the way humans sort cards, moving each element into its correct position step by step.
- In the given problem, since wallets are already sorted, insertion sort will perform minimal operations (only checking without swapping), making it the best fit.

Code:

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

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one step ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int wallets[20];

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

    int n = 20;

    cout << "Wallets before sorting:\n";
    for (int i = 0; i < n; i++)
        cout << wallets[i] << " ";
    cout << endl;
    // Applying Insertion Sort
    insertionSort(wallets, n);

    cout << "Wallets after sorting:\n";
    for (int i = 0; i < n; i++)
        cout << wallets[i] << " ";
    cout << endl;

    return 0;
}
```

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.**

- Counting Sort is best when the maximum value in the dataset is small**,** because it directly counts occurrences of each value instead of comparing repeatedly**.**
- It is more efficient than comparison-based algorithms (like insertion sort or quicksort) when the data range (0–6 here) is limited and smaller than the number of elements.
- For this wallet problem, Counting Sort will quickly arrange wallets by counting money values (0 to 6) and then reconstructing the sorted list.

Code:

```
#include <iostream>

using namespace std;

 void countingSort(int arr[], int n, int maxVal) {

    int count[maxVal + 1] = {0};



    // Count frequency of each value

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

        count[arr[i]]++;



    // Place values back in sorted order

    int index = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}

int main() {
    int wallets[10] = {2, 3, 0, 2, 1, 4, 0, 2, 3, 6};
    int n = 10;
    int maxVal = 6;

    cout << "Wallets before sorting:\n";
    for (int i = 0; i < n; i++)
        cout << wallets[i] << " ";
    cout << endl;

    countingSort(wallets, n, maxVal);

    cout << "Wallets after sorting:\n";
    for (int i = 0; i < n; i++)
        cout << wallets[i] << " ";
    cout << endl;
    return 0;

  }
```

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?

- Efficient for larger unsorted data with wide value ranges because it uses the divide-and-conquer approach.
- Performs faster on average compared to many other sorting algorithms like insertion or selection sort.
- Good choice for competitive settings (like this fest), as it works well even when the data is random.

  Code:

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

// Function to swap two numbers
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);
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);   // before pivot
        quickSort(arr, pi + 1, high);  // after pivot
    }
}

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

    cout << "Scores before sorting:\n";
    for (int i = 0; i < n; i++)
        cout << scores[i] << " ";
    cout << endl;

    quickSort(scores, 0, n - 1);

    cout << "Scores after sorting (ascending):\n";
    for (int i = 0; i < n; i++)
        cout << scores[i] << " ";
    cout << endl;

    return 0;
}
```

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.

```
#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]; // 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);
        quickSort(arr, low, pi - 1);   // left side of pivot
        quickSort(arr, pi + 1, high);  // right side of pivot
    }
}

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

    cout << "Deadlines before sorting:\n";
    for (int i = 0; i < n; i++)
        cout << deadlines[i] << " ";
    cout << endl;

    quickSort(deadlines, 0, n - 1);

    cout << "Deadlines after sorting (ascending):\n";
    for (int i = 0; i < n; i++)
        cout << deadlines[i] << " ";
    cout << endl;

    return 0;
}
```

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.

- Each time we connect the midpoints of a square, the new square has **half the area** of the previous one.
  Example: If SQ-1 has area `A`, then SQ-2 = `A/2`, SQ-3 = `A/4`, and so on.
- So, the areas form a **sorted sequence in descending order**.

- Since the array is sorted, the **best searching technique is Binary Search** (fast and efficient with O(log n) complexity).

Code:

```
#include <iostream>
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 (arr[mid] == key)
            return true;
        else if (arr[mid] > key)
            low = mid + 1;   // move right
        else
            high = mid - 1;  // move left
    }
    return false;
}

int main() {
    double areas[50];
    double initialArea;

    cout << "Enter the area of SQ-1: ";
    cin >> initialArea;

    // Generate areas of 50 squares
    areas[0] = initialArea;
    for (int i = 1; i < 50; i++) {
        areas[i] = areas[i - 1] / 2.0;  // each next is half
    }

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

    if (binarySearch(areas, 50, key))
        cout << "Area found in the array." << endl;
    else
        cout << "Area not found in the array." << endl;
    return 0;
}
```

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?
    1. In this activity, each player introduces the next player. This is exactly like a linked list, where each node (player) stores information and a pointer to the next node (next player).
- How it Works?
    1. The head coach introduces the first player → this is the head node.
    2. Each player introduces the next player → this is the link (pointer to next node).
    3. The process continues until no next player exists → this is when we reach the last node (NULL).
- Traversal (Meeting Players):
    1. The chief guest moves forward node by node in the linked list, meeting every player sequentially until all are covered.

Code:

```
#include <iostream>

using namespace std;



// Node structure for each player

struct Player {

    string name;

    Player* next;

};



// Function to create a new player node

Player* createPlayer(string name) {

    Player* newPlayer = new Player;

    newPlayer->name = name;
    newPlayer->next = NULL;
    return newPlayer;
}

// Function to display introductions
void introducePlayers(Player* head) {
    Player* current = head;
    cout << "Chief Guest starts meeting players:\n";
    while (current != NULL) {
        cout << "Meeting player: " << current->name << endl;
        current = current->next; // move to next player
    }
}

int main() {
    // Creating linked list of players
    Player* head = createPlayer("Player 1");
    head->next = createPlayer("Player 2");
    head->next->next = createPlayer("Player 3");
    head->next->next->next = createPlayer("Player 4");
    head->next->next->next->next = createPlayer("Player 5");

    // Chief guest meets players
    introducePlayers(head);
```

```
   return 0;

}
```

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.**

Code:

```
        #include <iostream>

using namespace std;



// Node structure for each bus stop

struct Stop {

    string name;

    Stop* next;

    Stop* prev;

};



// Function to create a new stop

Stop* createStop(string name) {

    Stop* newStop = new Stop;

    newStop->name = name;

    newStop->next = NULL;

    newStop->prev = NULL;

    return newStop;

}
```

```
// Traverse forward (Onward journey)

void traverseForward(Stop* head) {

    cout << "Onward Journey: ";

    Stop* current = head;

    while (current != NULL) {

        cout << current->name;

        if (current->next != NULL) cout << " -> ";

        current = current->next;

    }

    cout << endl;

}


// Traverse backward (Return journey)

void traverseBackward(Stop* tail) {

    cout << "Return Journey: ";

    Stop* current = tail;

    while (current != NULL) {

        cout << current->name;

        if (current->prev != NULL) cout << " -> ";

        current = current->prev;

    }

    cout << endl;

}
```

```
int main() {

    // Create stops

    Stop* A = createStop("A");

    Stop* B = createStop("B");

    Stop* C = createStop("C");

    Stop* D = createStop("D");


    // Link them forward

    A->next = B;

    B->prev = A;

    B->next = C;

    C->prev = B;

    C->next = D;

    D->prev = C;


    // Traversals

    traverseForward(A); // start from head (A)

    traverseBackward(D); // start from tail (D)


    return 0;

}
```

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**

Code:

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

int main() {
    // Dalta Gang: 4x2 matrix
    int dalta[4][2] = {
        {1, 2},
        {3, 4},
        {5, 6},
        {7, 8}
    };

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

    // Result matrix: 4x3
    int result[4][3] = {0};

    // Matrix multiplication
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3; j++) {
```

```
        result[i][j] = 0;

        for (int k = 0; k < 2; k++) {

            result[i][j] += dalta[i][k] * malta[k][j];

        }

    }

}


// Display Dalta Gang matrix

cout << "Dalta Gang Matrix (4x2):\n";

for (int i = 0; i < 4; i++) {

    for (int j = 0; j < 2; j++)

        cout << dalta[i][j] << " ";

    cout << endl;

}


// Display Malta Gang matrix

cout << "\nMalta Gang Matrix (2x3):\n";

for (int i = 0; i < 2; i++) {

    for (int j = 0; j < 3; j++)

        cout << malta[i][j] << " ";

    cout << endl;

}


// Display Result

cout << "\nResultant Matrix (4x3):\n";
```

```
    for (int i = 0; i < 4; i++) {

        for (int j = 0; j < 3; j++)

            cout << result[i][j] << " ";

        cout << endl;

    }



    return 0;

}
```

DSA LAB QUESTIONS

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* createNode(string name) {

   Node* n = new Node;

   n->name = name;

   n->left = n->right = NULL;

   return n;

}


Node* insert(Node* root, string name) {

   if (root == NULL) return createNode(name);
```

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

Node* minNode(Node* node) {
    while (node && node->left != NULL) node = node->left;
    return node;
}

Node* search(Node* root, string key) {
    if (root == NULL || root->name == key) return root;
    if (key < root->name) return search(root->left, key);
    else return search(root->right, key);
}

Node* successor(Node* root, Node* target) {
    if (target->right != NULL) return minNode(target->right);
    Node* succ = NULL;
    while (root != NULL) {
        if (target->name < root->name) {
            succ = root;
            root = root->left;
        } else if (target->name > root->name) {
            root = root->right;
        } else break;
    }
    return succ;
}

int main() {
    Node* root = NULL;
    string names[] = {"Q","S","R","T","M","A","B","P","N"};
    for (string x : names) root = insert(root, x);

    Node* target = search(root, "M");
    Node* succ = successor(root, target);

    if (succ) cout << "Successor of M is: " << succ->name << endl;
    else cout << "No successor." << endl;
    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;

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

Node* createNode(int val) {
    Node* n = new Node;
    n->data = val;
    n->left = n->right = NULL;
    return n;
}

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

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

void preorder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

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

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

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

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

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

Node* createNode(int v) {
    Node* n = new Node;
    n->data = v;
    n->left = n->right = NULL;
    return n;
}

Node* insert(Node* root, int v) {
    if (root == NULL) return createNode(v);
    if (v < root->data) root->left = insert(root->left, v);
    else if (v > root->data) root->right = insert(root->right, v);
    return root;
}

Node* search(Node* root, int key) {
    if (root == NULL || root->data == key) return root;
    if (key < root->data) return search(root->left, key);
    else return search(root->right, key);
}

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

    int key;
    cout << "Enter key to search: ";
    cin >> key;
    Node* res = search(root, key);
    if (res) cout << key << " found" << endl;
    else cout << key << " not found" << endl;
    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;

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

Node* createNode(int v) {
    Node* n = new Node;
    n->data = v;
    n->left = n->right = NULL;
    return n;
}
```

```
Node* insert(Node* root, int v) {
    if (root == NULL) return createNode(v);
    if (v < root->data) root->left = insert(root->left, v);
    else if (v > root->data) root->right = insert(root->right, v);
    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 = NULL;
    for (int r: rolls) root = insert(root, r);

    cout << "Ascending order: ";
    inorder(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* createNode(int v) {
   Node* n = new Node;
   n->data = v;
   n->left = n->right = NULL;
   return n;
}

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

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

```
    return root;
}

Node* deleteNode(Node* root, int key) {
    if (!root) return root;
    if (key < root->data) root->left = deleteNode(root->left, key);
    else if (key > root->data) root->right = deleteNode(root->right, key);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

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

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

    cout << "Before: "; inorder(root); cout << endl;

    int key; cout << "Delete which node? "; cin >> key;
    root = deleteNode(root, key);

    cout << "After: "; 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* createNode(string s) {
    Node* n = new Node;
    n->name = s;
    n->left = n->right = NULL;
    return n;
}

Node* insert(Node* root, string s) {
    if (!root) return createNode(s);
    if (s < root->name) root->left = insert(root->left, s);
    else if (s > root->name) root->right = insert(root->right, s);
    return root;
}

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

Node* search(Node* root, string key) {
    if (!root || root->name == key) return root;
    if (key < root->name) return search(root->left, key);
    else return search(root->right, key);
}

int main() {
    string members[] =
{"John","Alice","Robert","Zara","Mary","David","Emily"};
    Node* root = NULL;
    for (string m: members) root = insert(root, m);

    cout << "Family (Inorder): ";
    inorder(root); cout << endl;

    string key; cout << "Enter name to search: ";
    getline(cin, key);
    if (search(root, key)) cout << key << " found." << endl;
    else cout << key << " not found." << endl;
    return 0;
}
```