Section: A

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.**

   Ans:

   Theory:
   The numbers are already sorted. Best is to check if sorted and not sort again. If sorting is required, Insertion Sort is best because it is very fast for sorted data.

   Code:

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

int main() {
   int a[20];
   for (int i = 0; i < 20; i++) a[i] = 20 + i*10;

   bool sorted = true;
   for (int i = 1; i < 20; i++) {
      if (a[i] < a[i-1]) { sorted = false; break; }
   }

   if (sorted) {
      cout << "Array is already sorted\n";
   } else {
      for (int i = 1; i < 20; i++) {
         int key = a[i];
         int j = i-1;
         while (j >= 0 && a[j] > key) {
            a[j+1] = a[j];
            j--;
```

```
            }
            a[j+1] = key;
        }
        cout << "Sorted array: ";
        for (int i = 0; i < 20; i++) cout << a[i] << " ";
    }
    return 0;
}
```

Explanation:
We first check if array is sorted. If yes, we do nothing. If no, we use insertion sort.

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.**

   Ans:

   Theory:
   When numbers are small and repeated, Counting Sort is best. It counts how many times each number appears and then prints them in order.

   Code:

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

int main() {
    int a[10] = {2,0,3,2,0,1,3,4,0,0};
    int maxVal = 6;
    int count[7] = {0};

    for (int i = 0; i < 10; i++) count[a[i]]++;

    cout << "Sorted wallets: ";
    for (int i = 0; i <= maxVal; i++) {
```

```
        for (int j = 0; j < count[i]; j++) {
            cout << i << " ";
        }
    }
    return 0;
}
```

Explanation:
We count each number and then print them. This gives sorted result quickly.

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.**

Ans:

Theory:
The numbers are random. Quick Sort is best because it works fast on such data.

Code:

```
#include <iostream>

using namespace std;


int partitionArr(int a[], int low, int high) {
    int pivot = a[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
```

```
        if (a[j] <= pivot) {
            i++;
            int temp = a[i]; a[i] = a[j]; a[j] = temp;
        }
    }
    int temp = a[i+1]; a[i+1] = a[high]; a[high] = temp;
    return i+1;
}

void quickSort(int a[], int low, int high) {
    if (low < high) {
        int pi = partitionArr(a, low, high);
        quickSort(a, low, pi-1);
        quickSort(a, pi+1, high);
    }
}

int main() {
    int scores[12] = {45,12,78,34,23,89,67,11,90,54,32,76};
    quickSort(scores, 0, 11);
    cout << "Sorted scores: ";
    for (int i = 0; i < 12; i++) cout << scores[i] << " ";
    return 0;
}
```

Explanation:
Quick sort selects a pivot, puts smaller numbers on left and larger on right, and repeats.

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

Ans:

Theory:
Divide-and-conquer with unequal parts is Quick Sort. It divides array into two parts and sorts.

Code:

```
#include <iostream>

using namespace std;


int partitionArr(int a[], int low, int high) {
   int pivot = a[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (a[j] <= pivot) {
         i++;
         int temp = a[i]; a[i] = a[j]; a[j] = temp;
      }
   }
   int temp = a[i+1]; a[i+1] = a[high]; a[high] = temp;
   return i+1;
```

```
}

void quickSort(int a[], int low, int high) {
   if (low < high) {
      int pi = partitionArr(a, low, high);
      quickSort(a, low, pi-1);
      quickSort(a, pi+1, high);
   }
}

int main() {
   int deadlines[10] = {25,12,45,7,30,18,40,22,10,35};
   quickSort(deadlines, 0, 9);
   cout << "Sorted deadlines: ";
   for (int i = 0; i < 10; i++) cout << deadlines[i] << " ";
   return 0;
}
```

Explanation:
Quick sort is used. After sorting, the nearest deadlines will appear first.

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.**

Ans:

Theory:
If data is sorted, the best search is Binary Search. It checks the middle, then goes left or right. It is faster than linear search.

Code:

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

int binarySearch(double a[], int n, double key) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (a[mid] == key) return mid;
        else if (a[mid] < key) l = mid + 1;
        else r = mid - 1;
    }
    return -1;
}

int main() {
    double areas[5] = {100, 80, 60, 40, 20}; // example values
    int n = 5;

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

```
    int pos = binarySearch(areas, n, key);
    if (pos != -1) cout << "Area found at position " << pos << endl;
    else cout << "Area not found" << endl;

    return 0;
}
```

Explanation:
We use binary search on the array of areas. It checks the middle element and reduces the search size each step.

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.**

Ans:

Theory:
A singly linked list is best. Each node stores a player name and a pointer to the next player.

Code:

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

struct Node {
    string name;
    Node* next;
```

```
    Node(string s) { name = s; next = NULL; }
};

int main() {
    string players[5] =
{"Player1","Player2","Player3","Player4","Player5"};

    Node* head = NULL;
    Node* tail = NULL;

    for (int i = 0; i < 5; i++) {
        Node* node = new Node(players[i]);
        if (head == NULL) head = tail = node;
        else {
            tail->next = node;
            tail = node;
        }
    }

    cout << "Chief guest meets players:\n";
    Node* cur = head;
    while (cur != NULL) {
        cout << cur->name << endl;
        cur = cur->next;
    }
    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.

   Theory:
   A doubly linked list allows forward and backward travel. Each node has next and prev pointers.

   Code:

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

struct Node {
   string stop;
   Node* next;
   Node* prev;
   Node(string s) { stop = s; next = prev = NULL; }
};

int main() {
   string stops[4] = {"A","B","C","D"};
   Node* head = NULL;
   Node* tail = NULL;

   for (int i = 0; i < 4; i++) {
      Node* node = new Node(stops[i]);
      if (head == NULL) head = tail = node;
      else {
         tail->next = node;
         node->prev = tail;
         tail = node;
      }
   }

   cout << "Onward journey: ";
   Node* cur = head;
   while (cur != NULL) {
```

```
        cout << cur->stop << " ";
        cur = cur->next;
    }
    cout << endl;

    cout << "Return journey: ";
    cur = tail;
    while (cur != NULL) {
        cout << cur->stop << " ";
        cur = cur->prev;
    }
    cout << endl;

    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

   Theory:
   Matrix multiplication: (4×2) × (2×3) = (4×3). Each cell is row × column multiplication.

   Code:

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

int main() {
    int Dalta[4][2] = {{10,20},{5,15},{12,8},{7,9}};
    int Malta[2][3] = {{2,3,4},{1,0,5}};
    int result[4][3] = {0};

    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];
            }
        }
    }

    cout << "Result 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;
}
```

Explanation:
We multiply each row of Dalta with each column of Malta to form a new 4×3 matrix.

Section: B

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: 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.**

Ans:

Theory:
In BST, successor means the next bigger element in in-order traversal. If node has right child → leftmost of right subtree. Else → go up.

Code:

```
#include <iostream>
```

```
using namespace std;

struct Node {
    string key;
    Node* left;
    Node* right;
    Node(string k) { key = k; left = right = NULL; }
};

Node* insert(Node* root, string k) {
    if (root == NULL) return new Node(k);
    if (k < root->key) root->left = insert(root->left, k);
    else root->right = insert(root->right, k);
    return root;
}

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

Node* inorderSuccessor(Node* root, Node* n) {
    if (n->right != NULL) return minValue(n->right);
    Node* succ = NULL;
    while (root != NULL) {
        if (n->key < root->key) {
            succ = root;
```

```
            root = root->left;
        } else if (n->key > root->key) {
            root = root->right;
        } else break;
    }
    return succ;
}

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

int main() {
    string arr[9] = {"Q","S","R","T","M","A","B","P","N"};
    Node* root = NULL;
    for (int i = 0; i < 9; i++) root = insert(root, arr[i]);

    Node* mnode = search(root, "M");
    Node* succ = inorderSuccessor(root, mnode);

    if (succ != NULL) cout << "Successor of M is: " << succ->key <<
endl;
    else cout << "No successor found" << endl;

    return 0;
}
```

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

Ans:

Theory:

- Inorder → Left, Root, Right (gives sorted order)
- Preorder → Root, Left, Right
- Postorder → Left, Right, Root

Code:

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

struct Node {
   int val;
   Node* left;
   Node* right;
   Node(int v) { val = v; left = right = NULL; }
};

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

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

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

```
}

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

int main() {
    int vals[7] = {50,30,70,20,40,60,80};
    Node* root = NULL;
    for (int i = 0; i < 7; i++) root = insert(root, vals[i]);

    cout << "Inorder: "; inorder(root); cout << endl;
    cout << "Preorder: "; preorder(root); cout << endl;
    cout << "Postorder: "; postorder(root); cout << endl;

    return 0;
}
```

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

Ans:

Theory:
In BST, to search:

- If value = root → found.
- If value < root → go left.
- If value > root → go right.
  This repeats until found or tree ends.

Code:

```
#include <iostream>

using namespace std;
```

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

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

bool search(Node* root, int key) {
    if (root == NULL) return false;
    if (root->val == key) return true;
    if (key < root->val) return search(root->left, key);
    else return search(root->right, key);
}

int main() {
    int vals[10] = {45,12,78,34,23,89,67,11,90,54};
    Node* root = NULL;
    for (int i = 0; i < 10; i++) root = insert(root, vals[i]);
```

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

    if (search(root, key)) cout << key << " found\n";
    else cout << key << " not found\n";

    return 0;
}
```

Explanation:
We insert values into BST. Search checks left or right until the number is found.

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.**

Ans:

Theory:
In-order traversal of BST prints values in ascending order.

Code:

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

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

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

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

int main() {
    int rolls[10] = {45,12,78,34,23,89,67,11,90,54};
    Node* root = NULL;
    for (int i = 0; i < 10; i++) root = insert(root, rolls[i]);

    cout << "In-order (ascending): ";
```

```
    inorder(root);
    cout << endl;

    return 0;
}
```

Explanation:
BST is built from roll numbers. In-order prints them in increasing order.

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.**

Ans:

Theory:
To delete from BST:

1. If node has no child → remove it.
2. If one child → replace with child.
3. If two children → replace with inorder successor, then delete successor.

Code:

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

struct Node {
```

```
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

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

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

Node* deleteNode(Node* root, int key) {
    if (root == NULL) return root;
    if (key < root->val) root->left = deleteNode(root->left, key);
    else if (key > root->val) root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            delete root;
            return temp;
```

```
        } else if (root->right == NULL) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* succ = minValue(root->right);
        root->val = succ->val;
        root->right = deleteNode(root->right, succ->val);
    }
    return root;
}

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

int main() {
    int arr[7] = {50,30,70,20,40,60,80};
    Node* root = NULL;
    for (int i = 0; i < 7; i++) root = insert(root, arr[i]);

    cout << "In-order before deletion: ";
    inorder(root); cout << endl;
```

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

    root = deleteNode(root, key);

    cout << "In-order after deletion: ";
    inorder(root); cout << endl;

    return 0;
}
```

Explanation:
We handle all three delete cases. After deletion, we print BST again using in-order.

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.

   Ans:

   Theory:
   We can store family members in BST by their names. Traversals give different views, and search finds a member quickly.

   Code:

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

```
struct Node {
    string name;
    Node* left;
    Node* right;
    Node(string s) { name = s; left = right = NULL; }
};

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

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

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

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

bool search(Node* root, string key) {
    if (root == NULL) return false;
    if (root->name == key) return true;
    if (key < root->name) return search(root->left, key);
```

```
    else return search(root->right, key);
}

int main() {
    string family[7] =
{"John","Alice","Bob","Mary","David","Zara","Peter"};
    Node* root = NULL;
    for (int i = 0; i < 7; i++) root = insert(root, family[i]);

    cout << "In-order: "; inorder(root); cout << endl;
    cout << "Pre-order: "; preorder(root); cout << endl;
    cout << "Post-order: "; postorder(root); cout << endl;

    string q;
    cout << "Enter name to search: ";
    cin >> q;
    if (search(root, q)) cout << q << " found\n";
    else cout << q << " not found\n";

    return 0;
}
```

Explanation:
We store family names in BST. We display tree in 3 traversals and check if a given name is in the tree.