Complete C++ Solutions: Section A & Section B (Each question = separate program)

SECTION - A

Q1: 20 wallets in a row (20,30,40,...). Best sort and program (array already sorted).

```
// Q1 - Bubble Sort (optimized) - array already sorted but algorithm checks for swaps
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; ++i) {
        swapped = false;
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break; // array already sorted
    }
}

int main() {
    const int N = 20;
    int arr[N];
    int val = 20;
    for (int i = 0; i < N; ++i) {
        arr[i] = val;
        val += 10;
    }

    cout << "Original wallets: ";
    for (int i = 0; i < N; ++i) cout << arr[i] << " ";
    cout << endl;

    bubbleSort(arr, N);

    cout << "After sorting (bubble sort): ";
    for (int i = 0; i < N; ++i) cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Q2: 10 wallets with values (max 6), use Counting Sort.

```
// Q2 - Counting Sort for small-range integers (0..6)
#include <iostream>
#include <vector>
using namespace std;

void countingSort(vector<int>& arr, int maxVal) {
    int n = arr.size();
    vector<int> count(maxVal + 1, 0);
    for (int x : arr) count[x]++;
    for (int i = 1; i <= maxVal; ++i) count[i] += count[i - 1];
    vector<int> output(n);
    for (int i = n - 1; i >= 0; --i) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }
    for (int i = 0; i < n; ++i) arr[i] = output[i];
}

int main() {
    vector<int> wallets = {2, 2, 3, 3, 0, 0, 2, 1, 4, 6}; // given distribution
    int maxVal = 6;

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

    countingSort(wallets, maxVal);

    cout << "Sorted wallets (counting sort): ";
    for (int v : wallets) cout << v << " ";
    cout << endl;
    return 0;
}
```

Q3: Quick Sort for 12 students' scores.

```
// Q3 - Quick Sort (ascending)
#include <iostream>
using namespace std;

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    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;
}

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);
    }
}

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

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

    quickSort(arr, 0, n - 1);

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

Q4: Sort project deadlines using divide-and-conquer (Quick Sort).

```
// Q4 - Quick Sort for deadlines
#include <iostream>
using namespace std;

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    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;
}

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);
    }
}

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

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

    quickSort(deadlines, 0, n - 1);

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

Q5: Search in array of areas for SQ-1..SQ-50 (each midpoint square area = half previous). Use Binary Search.

```
// Q5 - Generate 50 squares by connecting midpoints (area halves each time), then binary search
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

bool binarySearch(const vector<double>& arr, double key) {
    int l = 0, r = (int)arr.size() - 1;
    const double EPS = 1e-9;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (fabs(arr[mid] - key) < EPS) return true;
        if (arr[mid] < key) l = mid + 1;
        else r = mid - 1;
    }
    return false;
}

int main() {
    // We'll take initial area input for SQ-1 from the user to be precise.
    double area1;
    cout << "Enter area of SQ-1 (positive number): ";
    if (!(cin >> area1) || area1 <= 0) {
        cout << "Invalid input. Using default area = 100.0" << endl;
        area1 = 100.0;
    }

    vector<double> areas(50);
    areas[0] = area1;
    for (int i = 1; i < 50; ++i) {
        areas[i] = areas[i - 1] / 2.0; // midpoint-square area = 1/2 of previous
    }

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

    if (binarySearch(areas, key)) cout << "Area found among SQ-1..SQ-50." << endl;
    else cout << "Area NOT found." << endl;

    // Optional: display generated areas (first 10 and last 3)
    cout << "Sample areas (first 10): ";
    for (int i = 0; i < 10; ++i) cout << areas[i] << " ";
    cout << endl;
    return 0;
}
```

Q6: Singly Linked List to model introductions (each player introduces next).

```
// Q6 - Singly Linked List for player introductions
#include <iostream>
#include <string>
using namespace std;

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

void append(Node*& head, const string& name) {
    Node* node = new Node(name);
    if (!head) { head = node; return; }
    Node* temp = head;
    while (temp->next) temp = temp->next;
    temp->next = node;
}

void introduceAll(Node* head) {
    Node* temp = head;
    while (temp) {
        cout << "Introducing: " << temp->name << endl;
        temp = temp->next;
    }
}

void cleanup(Node*& head) {
    while (head) {
        Node* t = head;
        head = head->next;
        delete t;
    }
}

int main() {
    Node* head = nullptr;
    int n;
    cout << "How many players? ";
    if (!(cin >> n) || n <= 0) {
        cout << "Invalid number. Exiting." << endl;
        return 0;
    }
    cin.ignore(); // consume newline
    for (int i = 0; i < n; ++i) {
        string name;
        cout << "Enter name of player " << i + 1 << ": ";
        getline(cin, name);
        append(head, name);
    }

    cout << "\nIntroductions sequence:" << endl;
    introduceAll(head);

    cleanup(head);
    return 0;
}
```

Q7: Doubly Linked List to model bus stops A->B->C->D and back.

```
// Q7 - Doubly Linked List for bus journey forward and backward
#include <iostream>
#include <string>
using namespace std;

struct Node {
    string stop;
    Node* next;
    Node* prev;
    Node(const string& s): stop(s), next(nullptr), prev(nullptr) {}
};

void append(Node*& head, const string& stop) {
    Node* node = new Node(stop);
    if (!head) { head = node; return; }
    Node* temp = head;
    while (temp->next) temp = temp->next;
    temp->next = node;
    node->prev = temp;
}

void traverseForward(Node* head) {
    Node* temp = head;
    cout << "Onward journey: ";
    while (temp) {
        cout << temp->stop;
        if (temp->next) cout << " -> ";
        temp = temp->next;
    }
    cout << endl;
}

void traverseBackward(Node* head) {
    if (!head) return;
    Node* tail = head;
    while (tail->next) tail = tail->next;
    cout << "Return journey: ";
    while (tail) {
        cout << tail->stop;
        if (tail->prev) cout << " -> ";
        tail = tail->prev;
    }
    cout << endl;
}

void cleanup(Node*& head) {
    while (head) {
        Node* t = head;
        head = head->next;
        delete t;
    }
}

int main() {
    Node* head = nullptr;
    // Store bus stops A -> B -> C -> D (can be extended)
    append(head, "A");
    append(head, "B");
    append(head, "C");
    append(head, "D");

    traverseForward(head);
    traverseBackward(head);

    cleanup(head);
    return 0;
}
```

Q8: Dalta Gang (4x2) multiplied by Malta Gang (2x3) - display and multiply.

```
// Q8 - Matrix display and multiplication (Dalta 4x2) * (Malta 2x3) = Result 4x3
#include <iostream>
using namespace std;

int main() {
    const int R1 = 4, C1 = 2, R2 = 2, C2 = 3;
    int Dalta[R1][C1] = { {10, 20}, {30, 40}, {50, 60}, {70, 80} }; // sample values
    int Malta[R2][C2] = { {1, 2, 3}, {4, 5, 6} }; // sample values

    cout << "Dalta Gang matrix (" << R1 << "x" << C1 << "):\n";
    for (int i = 0; i < R1; ++i) {
        for (int j = 0; j < C1; ++j) cout << Dalta[i][j] << " ";
        cout << endl;
    }

    cout << "\nMalta Gang matrix (" << R2 << "x" << C2 << "):\n";
    for (int i = 0; i < R2; ++i) {
        for (int j = 0; j < C2; ++j) cout << Malta[i][j] << " ";
        cout << endl;
    }

    int Result[R1][C2] = {0};
    for (int i = 0; i < R1; ++i) {
        for (int j = 0; j < C2; ++j) {
            for (int k = 0; k < C1; ++k) {
                Result[i][j] += Dalta[i][k] * Malta[k][j];
            }
        }
    }

    cout << "\nResult matrix (Dalta x Malta) (" << R1 << "x" << C2 << "):\n";
    for (int i = 0; i < R1; ++i) {
        for (int j = 0; j < C2; ++j) cout << Result[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

SECTION - B

Q1: Create BST with letters {Q,S,R,T,M,A,B,P,N} and find successor of 'M'.

```
// B1 - BST with chars and find inorder successor of 'M'
#include <iostream>
using namespace std;

struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char c): data(c), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, char 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* minValue(Node* node) {
    Node* cur = node;
    while (cur && cur->left) cur = cur->left;
    return cur;
}

// Find inorder successor of key in BST rooted at root.
// If node has right subtree -> min of right subtree.
// Otherwise track ancestor where we took left turn.
Node* inorderSuccessor(Node* root, char key) {
    Node* succ = nullptr;
    Node* cur = root;
    while (cur) {
        if (key < cur->data) {
            succ = cur;
            cur = cur->left;
        } else if (key > cur->data) {
            cur = cur->right;
        } else {
            if (cur->right) succ = minValue(cur->right);
            break;
        }
    }
    return succ;
}

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

int main() {
    char vals[] = {'Q','S','R','T','M','A','B','P','N'};
    Node* root = nullptr;
    for (char c : vals) root = insert(root, c);

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

    char key = 'M';
    Node* succ = inorderSuccessor(root, key);
    if (succ) cout << "Inorder successor of '" << key << "' is '" << succ->data << "'.
";
    else cout << "No inorder successor found for '" << key << "'.
";
    return 0;
}
```

Q2: Implement Inorder, Preorder, Postorder traversals of BST.

```
// B2 - BST traversals (inorder, preorder, postorder). Demonstration with sample numbers.
#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);
}
void preorder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
void postorder(Node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

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

    cout << "Inorder: ";
    inorder(root);
    cout << "\nPreorder: ";
    preorder(root);
    cout << "\nPostorder: ";
    postorder(root);
    cout << endl;
    return 0;
}
```

Q3: Search an element in a given BST.

```
// B3 - Search in BST (iterative and recursive versions). Using integer 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;
}

Node* searchBST(Node* root, int key) {
    if (!root) return nullptr;
    if (root->data == key) return root;
    if (key < root->data) return searchBST(root->left, key);
    return searchBST(root->right, key);
}

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

    int key;
    cout << "Enter value to search: ";
    cin >> key;
    Node* found = searchBST(root, key);
    if (found) cout << "Value " << key << " found in BST.
";
    else cout << "Value " << key << " NOT found in BST.
";
    return 0;
}
```

Q4: Construct BST from roll numbers and perform in-order traversal (ascending).

```
// B4 - Build BST from given roll numbers and print 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 v : rolls) root = insert(root, v);

    cout << "Roll numbers in ascending order (inorder traversal):
";
    inorder(root);
    cout << endl;
    return 0;
}
```

Q5: Delete a node from BST (rolls: 50,30,70,20,40,60,80).

```
// B5 - Delete a node from BST (handles 0,1,2 child cases)
#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* minValueNode(Node* node) {
    Node* cur = node;
    while (cur && cur->left) cur = cur->left;
    return cur;
}

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 {
        // node found
        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 = minValueNode(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 arr[] = {50, 30, 70, 20, 40, 60, 80};
    Node* root = nullptr;
    for (int v : arr) root = insert(root, v);

    cout << "Initial BST (inorder): ";
    inorder(root); cout << endl;

    int key;
    cout << "Enter roll number to delete: ";
    cin >> key;
    root = deleteNode(root, key);

    cout << "BST after deletion (inorder): ";
    inorder(root); cout << endl;
    return 0;
}
```

Q6: Family tree hierarchy using BST (insert names, traversals, search).

```
// B6 - Family tree using BST keyed by name (string). Supports insert, traversals, and 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 root->right = insert(root->right, name);
    return root;
}

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

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

int main() {
    Node* root = nullptr;
    // Sample family members (you can modify / input)
    string members[] = {"Mohan", "Anita", "Rahul", "Sunita", "Pooja", "Neha", "Ramesh"};
    for (const string& s : members) root = insert(root, s);

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

    cout << "Preorder: ";
    preorder(root); cout << endl;

    cout << "Postorder: ";
    postorder(root); cout << endl;

    string query;
    cout << "Enter name to search: ";
    getline(cin, query);
    if (query.empty()) getline(cin, query); // handle newline
    Node* found = search(root, query);
    if (found) cout << query << " found in family tree.
";
    else cout << query << " NOT found in family tree.
";
    return 0;
}
```