I can't generate a PDF directly. However, I can provide the complete answers in a text format that you can easily copy and paste into a word processor to save as a PDF. The response is divided into sections, with a brief explanation and a corresponding C++ program for each question.

Section A: Sorting, Linked Lists, and 2D Arrays

Q1: Wallet Sorting

The wallets are already sorted in ascending order. If sorting were necessary,

Insertion Sort would be the best choice. This is because it's efficient for data that is already sorted or nearly sorted, requiring fewer comparisons and swaps.

C++

```
#include <iostream>

#include <vector>

#include <numeric>


using namespace std;


// Function to perform Insertion Sort

void insertionSort(vector<int>& arr) {

   int n = arr.size();

   for (int i = 1; i < n; ++i) {

      int key = arr[i];

      int j = i - 1;

      while (j >= 0 && arr[j] > key) {

         arr[j + 1] = arr[j];

         j = j - 1;

      }

      arr[j + 1] = key;

   }

}


int main() {

   vector<int> wallets(20);

   for (int i = 0; i < wallets.size(); ++i) {

      wallets[i] = 20 + i * 10;

   }


   cout << "Original wallet values: ";

   for (int val : wallets) {

      cout << val << " ";

   }

   cout << endl;
```

```
    insertionSort(wallets);


    cout << "Wallet values after sorting: ";
    for (int val : wallets) {
        cout << val << " ";
    }
    cout << endl;


    return 0;
}
```

<hr>

Q2: Sorting Wallets with a Limited Range

The best sorting technique for this scenario is

Counting Sort. The maximum amount in any wallet is $6, which means the data has a small and limited range. Counting Sort is a non-comparison-based algorithm that's highly efficient for sorting integers within a small range.

C++

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


using namespace std;


void countingSort(vector<int>& arr) {
    int max_val = 6;
    vector<int> count(max_val + 1, 0);
    for (int num : arr) {
        count[num]++;
    }


    vector<int> output(arr.size());
    int index = 0;
    for (int i = 0; i <= max_val; ++i) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
```

```
}

int main() {
   vector<int> wallets = {2, 2, 2, 3, 3, 0, 0, 1, 4};
   cout << "Original wallet values: ";
   for (int val : wallets) {
      cout << val << " ";
   }
   cout << endl;

   countingSort(wallets);

   cout << "Wallet values after sorting: ";
   for (int val : wallets) {
      cout << val << " ";
   }
   cout << endl;

   return 0;
}
```

<hr>

Q3: Student Score Sorting

Quick Sort is the most efficient technique to sort the scores. The problem specifies that the scores are unsorted and spread across a wide range, which are ideal conditions for a divide-and-conquer algorithm like Quick Sort.

C++

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

using namespace std;

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

   quickSort(scores, 0, scores.size() - 1);

   cout << "Scores after sorting: ";
   for (int score : scores) {
      cout << score << " ";
   }
   cout << endl;

   return 0;
}
```

<hr>

Q4: Project Deadline Sorting

The prompt hints at a

divide-and-conquer technique that divides the array into unequal parts. This is a defining characteristic of

Quick Sort, which partitions the array around a pivot, typically creating unequal sub-arrays.

C++

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

using namespace std;
```

```
int partition(vector<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(vector<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() {
    vector<int> deadlines = {25, 12, 45, 7, 30, 18, 40, 22, 10, 35};
    cout << "Original deadlines: ";
    for (int deadline : deadlines) {
        cout << deadline << " ";
    }
    cout << endl;

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

    cout << "Deadlines after sorting: ";
    for (int deadline : deadlines) {
        cout << deadline << " ";
    }
    cout << endl;

    return 0;
}
```

<hr>

Q5: Searching for an Area in an Array

The areas of the squares form a sorted sequence (in descending order), making

Binary Search the most efficient searching approach. Binary Search is designed for sorted data and has a time complexity of

O(logn), which is significantly faster than linear search for a large number of squares.

C++

```cpp
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

bool binarySearch(const vector<double>& arr, double target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (abs(arr[mid] - target) < 1e-9) {
            return true;
        } else if (arr[mid] > target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}

int main() {
    vector<double> areas;
    double initialArea = 100.0;
    for (int i = 0; i < 50; ++i) {
        areas.push_back(initialArea * pow(0.5, i));
    }

    double searchArea;
    cout << "Enter the area to search for: ";
    cin >> searchArea;

    if (binarySearch(areas, searchArea)) {
        cout << "The area " << searchArea << " is present in the array." << endl;
    } else {
        cout << "The area " << searchArea << " is not present in the array." << endl;
    }

    return 0;
}
```

<hr>

Q6: Player Introduction with Linked List

The process of one player introducing the next is a sequential, one-way relationship that can be perfectly modeled using a **Singly Linked List**. The chief guest's movement from one player to the next is analogous to traversing the list from head to tail.

C++

```
#include <iostream>
#include <string>

using namespace std;

struct PlayerNode {
    string name;
    PlayerNode* next;
    PlayerNode(string n) : name(n), next(nullptr) {}
};

class PlayerList {
private:
    PlayerNode* head;
public:
    PlayerList() : head(nullptr) {}
    void addPlayer(string name) {
        PlayerNode* newNode = new PlayerNode(name);
        if (!head) {
            head = newNode;
        } else {
            PlayerNode* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
        }
    }
    void introducePlayers() {
        PlayerNode* current = head;
        cout << "Chief guest meeting the players:" << endl;
        while (current) {
            cout << "Introducing " << current->name << endl;
            current = current->next;
        }
    }
    ~PlayerList() {
        PlayerNode* current = head;
        while (current) {
            PlayerNode* next = current->next;
            delete current;
            current = next;
        }
    }
};


int main() {
    PlayerList players;
    players.addPlayer("Alice");
    players.addPlayer("Bob");
    players.addPlayer("Charlie");
    players.addPlayer("David");
    players.introducePlayers();
    return 0;
}
```

<hr>

Q7: College Bus Journey with Doubly Linked List

The bus journey from A to D and back to A requires

bidirectional traversal, which is a core feature of a **Doubly Linked List**. Each bus stop is a node with pointers to both the next and the previous stop, enabling traversal in both forward and reverse directions.

C++

```
#include <iostream>
#include <string>
```

```
using namespace std;

struct BusStopNode {
   string stopName;
   BusStopNode* prev;
   BusStopNode* next;
   BusStopNode(string name) : stopName(name), prev(nullptr), next(nullptr) {}
};

class BusRoute {
private:
   BusStopNode* head;
   BusStopNode* tail;
public:
   BusRoute() : head(nullptr), tail(nullptr) {}
   void addStop(string name) {
      BusStopNode* newNode = new BusStopNode(name);
      if (!head) {
         head = newNode;
         tail = newNode;
      } else {
         tail->next = newNode;
         newNode->prev = tail;
         tail = newNode;
      }
   }
   void traverseForward() {
      BusStopNode* current = head;
      cout << "Onward Journey: ";
      while (current) {
         cout << current->stopName;
         if (current->next) {
            cout << " -> ";
         }
         current = current->next;
      }
      cout << endl;
   }
   void traverseBackward() {
        BusStopNode* current = tail;
        cout << "Return Journey: ";
        while (current) {
            cout << current->stopName;
            if (current->prev) {
                cout << " -> ";
            }
            current = current->prev;
        }
        cout << endl;
    }
    ~BusRoute() {
        BusStopNode* current = head;
        while (current) {
            BusStopNode* next = current->next;
            delete current;
            current = next;
        }
    }
};

int main() {
    BusRoute busRoute;
    busRoute.addStop("Stop A");
    busRoute.addStop("Stop B");
    busRoute.addStop("Stop C");
    busRoute.addStop("Stop D");
    busRoute.traverseForward();
    busRoute.traverseBackward();
    return 0;
}
```

<hr>

Q8: Matrix Operations

This problem requires storing the money values in 2D arrays and then performing matrix multiplication. The Dalta Gang matrix is 4x2 and the Malta Gang matrix is 2x3. Matrix multiplication is possible since the inner dimensions match.

C++

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

using namespace std;
```

```cpp
void displayMatrix(const vector<vector<int>>& matrix, const string& name) {
    cout << "Stored data for " << name << " in matrix form:" << endl;
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << "\t";
        }
        cout << endl;
    }
}

vector<vector<int>> multiplyMatrices(const vector<vector<int>>& A, const vector<vector<int>>& B) {
    int rows1 = A.size();
    int cols1 = A[0].size();
    int rows2 = B.size();
    int cols2 = B[0].size();
    if (cols1 != rows2) {
        return {};
    }
    vector<vector<int>> result(rows1, vector<int>(cols2, 0));
    for (int i = 0; i < rows1; ++i) {
        for (int j = 0; j < cols2; ++j) {
            for (int k = 0; k < cols1; ++k) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

int main() {
    vector<vector<int>> daltaGang = {{10, 20}, {15, 25}, {5, 30}, {22, 18}};
    vector<vector<int>> maltaGang = {{5, 10, 15}, {8, 12, 16}};

    displayMatrix(daltaGang, "Dalta Gang");
    cout << endl;
    displayMatrix(maltaGang, "Malta Gang");
    cout << endl;
    vector<vector<int>> resultMatrix = multiplyMatrices(daltaGang, maltaGang);
    if (!resultMatrix.empty()) {
        displayMatrix(resultMatrix, "Result of Dalta Gang * Malta Gang");
    }

    return 0;
}
```

<hr>

Section B: Binary Search Trees

Q1: Family Member BST and Successor

A

Binary Search Tree (BST) is an ideal way to store names for efficient searching, traversal, and insertion. The BST is built using the provided names. To find the

successor of the node M, the program searches for the smallest node whose value is greater than M.

C++

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

using namespace std;

struct TreeNode {
    string name;
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
    TreeNode(string n) : name(n), left(nullptr), right(nullptr), parent(nullptr) {}
};

TreeNode* insert(TreeNode* root, string name, TreeNode* parent = nullptr) {
    if (!root) {
        TreeNode* newNode = new TreeNode(name);
        newNode->parent = parent;
        return newNode;
    }
    if (name < root->name) {
        root->left = insert(root->left, name, root);
    } else if (name > root->name) {
        root->right = insert(root->right, name, root);
    }
```

```
  return root;
}

TreeNode* findNode(TreeNode* root, string name) {
  while (root && root->name != name) {
    root = (name < root->name) ? root->left : root->right;
  }
  return root;
}

TreeNode* findSuccessor(TreeNode* node) {
  if (!node) return nullptr;
  if (node->right) {
    TreeNode* current = node->right;
    while (current->left) {
      current = current->left;
    }
    return current;
  }
  TreeNode* current = node;
  TreeNode* parent = current->parent;
  while (parent && current == parent->right) {
    current = parent;
    parent = parent->parent;
  }
  return parent;
}

int main() {
  vector<string> names = {"Q", "S", "R", "T", "M", "A", "B", "P", "N"};
  TreeNode* root = nullptr;
  for (const string& name : names) {
    root = insert(root, name);
  }
  TreeNode* nodeM = findNode(root, "M");
  if (nodeM) {
    TreeNode* successor = findSuccessor(nodeM);
    if (successor) {
      cout << "The successor of M is: " << successor->name << endl;
```

```
        } else {
            cout << "M has no successor." << endl;
        }
    } else {
        cout << "Node M not found." << endl;
    }
    return 0;
}
```

<hr>

Q2: In-Order, Pre-Order, and Post-Order Traversal

This program implements the three standard traversal methods for a Binary Search Tree.

In-Order traversal visits the nodes in ascending order. **Pre-Order** traversal visits the root first, and **Post-Order** visits the root last.

C++

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

using namespace std;

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

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

void inOrderTraversal(TreeNode* root) {
    if (root) {
        inOrderTraversal(root->left);
        cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}
```

```
void preOrderTraversal(TreeNode* root) {
    if (root) {
        cout << root->data << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

void postOrderTraversal(TreeNode* root) {
    if (root) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        cout << root->data << " ";
    }
}

int main() {
    TreeNode* root = nullptr;
    vector<int> data = {50, 30, 70, 20, 40, 60, 80};
    for (int val : data) root = insert(root, val);

    cout << "In-Order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

<hr>

Q3: Searching an Element in a BST

A Binary Search Tree provides an efficient way to search for an element. The search function traverses the tree by comparing the target value with the current node's value and moving to the left or right child accordingly.

C++

```
#include <iostream>

using namespace std;

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

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

bool search(TreeNode* root, int target) {
   if (!root) return false;
   if (root->data == target) return true;
   if (target < root->data) return search(root->left, target);
   return search(root->right, target);
}

int main() {
   TreeNode* root = nullptr;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 70);
   insert(root, 20);
   insert(root, 40);

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

   if (search(root, target)) {
       cout << "Element " << target << " found." << endl;
   } else {
       cout << "Element " << target << " not found." << endl;
   }

   return 0;
}
```

<hr>

Q4: Constructing a BST for Roll Numbers

The university wants to store roll numbers for

fast searching, insertion, and retrieval in ascending order. A

Binary Search Tree is the perfect data structure for this. An

in-order traversal of the constructed BST will naturally display the roll numbers in ascending order.

C++

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

using namespace std;

struct TreeNode {
    int rollNumber;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int rn) : rollNumber(rn), left(nullptr), right(nullptr) {}
};

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

void inOrderTraversal(TreeNode* root) {
    if (root) {
        inOrderTraversal(root->left);
        cout << root->rollNumber << " ";
        inOrderTraversal(root->right);
    }
}

int main() {
    vector<int> rollNumbers = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};
    TreeNode* root = nullptr;
    cout << "Constructing BST..." << endl;
    for (int rn : rollNumbers) {
        root = insert(root, rn);
    }
    cout << "Roll numbers in ascending order (In-order traversal):" << endl;
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

<hr>

Q5: Deleting a Node from a BST

Deleting a student record from a BST is a common operation. This program implements a function to delete a node, handling all three cases: a node with no children, a node with one child, and a node with two children.

C++

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

using namespace std;

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

TreeNode* findMin(TreeNode* node) {
   while (node && node->left) {
      node = node->left;
   }
   return node;
}

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

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

void inOrderTraversal(TreeNode* root) {
   if (root) {
      inOrderTraversal(root->left);
      cout << root->data << " ";
      inOrderTraversal(root->right);
   }
}

int main() {
   vector<int> rollNumbers = {50, 30, 70, 20, 40, 60, 80};
   TreeNode* root = nullptr;
   for (int rn : rollNumbers) {
      root = insert(root, rn);

   }

   cout << "Original BST (In-order): ";

   inOrderTraversal(root);

   cout << endl;

   int keyToDelete;

   cout << "Enter student roll number to delete: ";

   cin >> keyToDelete;

   root = deleteNode(root, keyToDelete);

   cout << "BST after deletion (In-order): ";

   inOrderTraversal(root);

   cout << endl;


   return 0;

}
```

<hr>

Q6: Family Tree Hierarchy using BST

A

Binary Search Tree can be used to model a family tree, allowing efficient storage, retrieval, and manipulation of members. The program will insert members, search for a specific member, and display the hierarchy using in-order, pre-order, and post-order traversals.

C++

```
#include <iostream>

#include <string>

#include <vector>


using namespace std;


struct FamilyMemberNode {

    string name;

    FamilyMemberNode* left;

    FamilyMemberNode* right;

    FamilyMemberNode(string n) : name(n), left(nullptr), right(nullptr) {}

};


FamilyMemberNode* insertMember(FamilyMemberNode* root, string name) {

    if (!root) return new FamilyMemberNode(name);

    if (name < root->name) root->left = insertMember(root->left, name);

    else if (name > root->name) root->right = insertMember(root->right, name);

    return root;
```

```
}

void inOrderTraversal(FamilyMemberNode* root) {
    if (root) {
        inOrderTraversal(root->left);
        cout << root->name << " ";
        inOrderTraversal(root->right);
    }
}

void preOrderTraversal(FamilyMemberNode* root) {
    if (root) {
        cout << root->name << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

void postOrderTraversal(FamilyMemberNode* root) {
    if (root) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        cout << root->name << " ";
    }
}

bool searchMember(FamilyMemberNode* root, string name) {
    while (root) {
        if (root->name == name) return true;
        root = (name < root->name) ? root->left : root->right;
    }
    return false;
}

int main() {
    FamilyMemberNode* familyTree = nullptr;
    vector<string> members = {"Grandfather", "Father", "Uncle", "Aunt", "Son", "Daughter"};
    for (const string& member : members) {
        familyTree = insertMember(familyTree, member);
    }

    cout << "Family Tree Hierarchy (In-order): ";
    inOrderTraversal(familyTree);
    cout << endl;

    cout << "Family Tree Hierarchy (Pre-order): ";
    preOrderTraversal(familyTree);
    cout << endl;

    cout << "Family Tree Hierarchy (Post-order): ";
    postOrderTraversal(familyTree);
    cout << endl;

    string memberToFind = "Son";
    if (searchMember(familyTree, memberToFind)) {
        cout << "Found " << memberToFind << " in the family tree." << endl;
    } else {
        cout << memberToFind << " not found in the family tree." << endl;
    }

    return 0;
}
```