1) StudentAttendence.cpp

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

using namespace std;
// Function for Linear Search
bool linearSearch(const vector<int>& arr, int key) {
    for (int rollNo : arr) {
        if (rollNo == key) {
            return true;
        }
    }
    return false;
}

// Function for Binary Search
bool binarySearch(const vector<int>& arr, int key) {
    int left = 0;
    int right = arr.size() - 1;

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

void displayRollNumbers(const vector<int>& arr) {
    cout << "Roll Numbers: ";
    for(int roll : arr) {
        cout << roll << " ";
    }
    cout << endl;
}

int main() {
    vector<int> randomRolls = {15, 8, 27, 4, 42, 19, 31};
    vector<int> sortedRolls = {4, 8, 15, 19, 27, 31, 42};
    int key;

    cout << "------ Linear Search on Unsorted Data ------" << endl;
    displayRollNumbers(randomRolls);
    cout << "Enter roll number to search: ";
    cin >> key;
    if (linearSearch(randomRolls, key)) {
```

```
        cout << "Student with roll number " << key << " attended the program." << endl;
    } else {
        cout << "Student with roll number " << key << " did NOT attend the program." << endl;
    }

    cout << "\n------ Binary Search on Sorted Data ------" << endl;
    displayRollNumbers(sortedRolls);
    cout << "Enter roll number to search: ";
    cin >> key;
    if (binarySearch(sortedRolls, key)) {
        cout << "Student with roll number " << key << " attended the program." << endl;
    } else {
        cout << "Student with roll number " << key << " did NOT attend the program." << endl;
    }

    return 0;
}
```

Output:

------ Linear Search on Unsorted Data ------
Roll Numbers: 15 8 27 4 42 19 31
Enter roll number to search: 19
Student with roll number 19 attended the program.

------ Binary Search on Sorted Data ------
Roll Numbers: 4 8 15 19 27 31 42
Enter roll number to search: 10
Student with roll number 10 did NOT attend the program.

2) sorting.cpp

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

using namespace std;

// Function for Selection Sort
void selectionSort(vector<float>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        swap(arr[min_idx], arr[i]);
    }
}

// Function for Bubble Sort
void bubbleSort(vector<float>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void displayScores(const vector<float>& arr) {
    for (float score : arr) {
        cout << score << "% ";
    }
    cout << endl;
}

void displayTopFive(const vector<float>& arr) {
    cout << "Top five scores are: ";
    int n = arr.size();
    for (int i = n - 1; i >= max(0, n - 5); --i) {
        cout << arr[i] << "% ";
    }
    cout << endl;
}

int main() {
    vector<float> percentages = {65.5, 92.0, 78.3, 88.9, 59.1, 95.7, 85.2};
    vector<float> percentages_bubble = percentages;
```

```
    cout << "Original Percentages: ";
    displayScores(percentages);
    cout << endl;

    cout << "------ Using Selection Sort ------" << endl;
    selectionSort(percentages);
    cout << "Sorted Percentages: ";
    displayScores(percentages);
    displayTopFive(percentages);
    cout << endl;

    cout << "------ Using Bubble Sort ------" << endl;
    cout << "Original Percentages: ";
    displayScores(percentages_bubble);
    bubbleSort(percentages_bubble);
    cout << "Sorted Percentages: ";
    displayScores(percentages_bubble);
    displayTopFive(percentages_bubble);

    return 0;
}
```

Output:

Original Percentages: 65.5% 92% 78.3% 88.9% 59.1% 95.7% 85.2%

------ Using Selection Sort ------
Sorted Percentages: 59.1% 65.5% 78.3% 85.2% 88.9% 92% 95.7%
Top five scores are: 95.7% 92% 88.9% 85.2% 78.3%

------ Using Bubble Sort ------
Original Percentages: 65.5% 92% 78.3% 88.9% 59.1% 95.7% 85.2%
Sorted Percentages: 59.1% 65.5% 78.3% 85.2% 88.9% 92% 95.7%
Top five scores are: 95.7% 92% 88.9% 85.2% 78.3%

3) Hashing.cpp

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

using namespace std;
#define TABLE_SIZE 10
struct HashEntry {
   long long phone;
   string name;
};
class TelephoneBook {
private:
   vector<HashEntry> table;
public:
   TelephoneBook() {
      table.resize(TABLE_SIZE);
      for(int i = 0; i < TABLE_SIZE; ++i) {
         table[i].phone = -1; // -1 indicates empty slot
      }
   }
int hashFunction(long long key) {
      return key % TABLE_SIZE;
   }
 // Linear Probing
   void insertLinear(long long phone, const string& name) {
      int index = hashFunction(phone);
      int i = 0;
      while (table[(index + i) % TABLE_SIZE].phone != -1) {
         i++;
         if (i == TABLE_SIZE) {
            cout << "Hash table is full!" << endl;
            return;
         }
      }
      table[(index + i) % TABLE_SIZE] = {phone, name};
      cout << "Inserted " << name << " using Linear Probing." << endl;
   }

   void searchLinear(long long phone) {
      int index = hashFunction(phone);
      int i = 0;
      while (i < TABLE_SIZE) {
         int curr_index = (index + i) % TABLE_SIZE;
         if (table[curr_index].phone == phone) {
            cout << "Found: " << table[curr_index].name << " -> " << table[curr_index].phone << endl;
            return;
         }
         if (table[curr_index].phone == -1) {
            break;
         }
         i++;
      }
```

```
        cout << "Client with phone number " << phone << " not found." << endl;
    }

    // Quadratic Probing
    void insertQuadratic(long long phone, const string& name) {
        int index = hashFunction(phone);
        int i = 0;
        while (table[(index + i*i) % TABLE_SIZE].phone != -1) {
            i++;
            if (i == TABLE_SIZE) {
                cout << "Hash table is full or could not find a slot!" << endl;
                return;
            }
        }
        table[(index + i*i) % TABLE_SIZE] = {phone, name};
        cout << "Inserted " << name << " using Quadratic Probing." << endl;
    }

    void searchQuadratic(long long phone) {
        int index = hashFunction(phone);
        int i = 0;
        while (i < TABLE_SIZE) {
            int curr_index = (index + i*i) % TABLE_SIZE;
             if (table[curr_index].phone == phone) {
                cout << "Found: " << table[curr_index].name << " -> " << table[curr_index].phone << endl;
                return;
            }
            if (table[curr_index].phone == -1) {
                break;
            }
            i++;
        }
        cout << "Client with phone number " << phone << " not found." << endl;
    }
};
int main() {
    cout << "------ Using Linear Probing ------" << endl;
    TelephoneBook bookLinear;
    bookLinear.insertLinear(9876543210, "Alice");
    bookLinear.insertLinear(9876543211, "Bob"); // Collision with index 1
    bookLinear.insertLinear(9876543201, "Charlie"); // Collision with index 1
    bookLinear.searchLinear(9876543211);
    bookLinear.searchLinear(9876543201);
    bookLinear.searchLinear(9999999999);

    cout << "\n------ Using Quadratic Probing ------" << endl;
    TelephoneBook bookQuadratic;
    bookQuadratic.insertQuadratic(9876543210, "David");
    bookQuadratic.insertQuadratic(9876543211, "Eve"); // Collision with index 1
    bookQuadratic.insertQuadratic(9876543201, "Frank"); // Collision with index 1
    bookQuadratic.searchQuadratic(9876543211);
    bookQuadratic.searchQuadratic(9876543201);
```

```
    return 0;
}
```

Output:

------ Using Linear Probing ------
Inserted Alice using Linear Probing.
Inserted Bob using Linear Probing.
Inserted Charlie using Linear Probing.
Found: Bob -> 9876543211
Found: Charlie -> 9876543201
Client with phone number 9999999999 not found.

------ Using Quadratic Probing ------
Inserted David using Quadratic Probing.
Inserted Eve using Quadratic Probing.
Inserted Frank using Quadratic Probing.
Found: Eve -> 9876543211
Found: Frank -> 9876543201

4) PRN.cpp

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

using namespace std;

struct Node {
   int prn;
   string name;
   Node* next;
};

class PinnacleClub {
private:
   Node *president, *secretary;

public:
   PinnacleClub() {
      president = new Node{1, "President", nullptr};
      secretary = new Node{99, "Secretary", nullptr};
      president->next = secretary;
   }

   void addMember(int prn, string name) {
      Node* newNode = new Node{prn, name, nullptr};
      Node* temp = president;
      while(temp->next != secretary) {
         temp = temp->next;
      }
      newNode->next = secretary;
      temp->next = newNode;
      cout << "Member " << name << " added." << endl;
   }

   void deleteMember(int prn) {
      if (prn == president->prn || prn == secretary->prn) {
         cout << "Cannot delete president or secretary." << endl;
         return;
      }
      Node *prev = president;
      Node *curr = president->next;
      while(curr != secretary && curr->prn != prn) {
         prev = curr;
         curr = curr->next;
      }
      if (curr != secretary) {
         prev->next = curr->next;
         cout << "Member with PRN " << curr->prn << " deleted." << endl;
         delete curr;
      } else {
         cout << "Member with PRN " << prn << " not found." << endl;
      }
   }
```

```
    void displayMembers() {
        Node* temp = president;
        cout << "\n--- Pinnacle Club Members ---" << endl;
        while(temp != nullptr) {
            cout << "PRN: " << temp->prn << ", Name: " << temp->name << endl;
            temp = temp->next;
        }
        cout << "---------------------------\n";
    }

    void computeTotal() {
        int count = 0;
        Node* temp = president;
        while(temp != nullptr) {
            count++;
            temp = temp->next;
        }
        cout << "Total number of members (including President & Secretary): " << count << endl;
    }

    static void concatenateLists(PinnacleClub &list1, PinnacleClub &list2) {
        Node* temp = list1.president;
        while(temp->next != list1.secretary) {
            temp = temp->next;
        }
        temp->next = list2.president->next; // Link end of list1 to start of list2 (after pres)
        delete list2.president; // remove duplicate president

        Node* temp2 = temp;
        while(temp2->next != list2.secretary) {
            temp2 = temp2->next;
        }
        list1.secretary = list2.secretary; // Update secretary

        cout << "Lists concatenated." << endl;
    }
};

int main() {
    PinnacleClub divisionA;
    divisionA.addMember(10, "Alice");
    divisionA.addMember(20, "Bob");
    divisionA.displayMembers();
    divisionA.computeTotal();
    divisionA.deleteMember(20);
    divisionA.displayMembers();

    cout << "\nCreating another list for concatenation:" << endl;
    PinnacleClub divisionB;
    divisionB.addMember(30, "Charlie");
    divisionB.displayMembers();
```

```
    PinnacleClub::concatenateLists(divisionA, divisionB);
    cout << "\nFinal Concatenated List:" << endl;
    divisionA.displayMembers();
    divisionA.computeTotal();

    return 0;
}
```

Output:

```
Member Alice added.
Member Bob added.

--- Pinnacle Club Members ---
PRN: 1, Name: President
PRN: 10, Name: Alice
PRN: 20, Name: Bob
PRN: 99, Name: Secretary
---------------------------
Total number of members (including President & Secretary): 4
Member with PRN 20 deleted.

--- Pinnacle Club Members ---
PRN: 1, Name: President
PRN: 10, Name: Alice
PRN: 99, Name: Secretary
---------------------------

Creating another list for concatenation:
Member Charlie added.

--- Pinnacle Club Members ---
PRN: 1, Name: President
PRN: 30, Name: Charlie
PRN: 99, Name: Secretary
---------------------------
Lists concatenated.

Final Concatenated List:

--- Pinnacle Club Members ---
PRN: 1, Name: President
PRN: 10, Name: Alice
PRN: 30, Name: Charlie
PRN: 99, Name: Secretary
---------------------------
Total number of members (including President & Secretary): 4
```

5) CinemaHall.cpp

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

using namespace std;

const int ROWS = 10;
const int SEATS_PER_ROW = 7;

struct Seat {
    int id;
    bool isBooked;
    Seat *next, *prev;
};

class Cinemax {
private:
    vector<Seat*> row_heads;

public:
    Cinemax() {
        row_heads.resize(ROWS, nullptr);
        for (int i = 0; i < ROWS; ++i) {
            Seat* head = nullptr;
            Seat* tail = nullptr;
            for (int j = 1; j <= SEATS_PER_ROW; ++j) {
                Seat* newSeat = new Seat{j, false, nullptr, tail};
                if (head == nullptr) {
                    head = newSeat;
                } else {
                    tail->next = newSeat;
                }
                tail = newSeat;
            }
            row_heads[i] = head;
        }
        // Random initial bookings
        bookSeat(1, 3); bookSeat(1, 4); bookSeat(5, 5);
    }

    void displaySeats() {
        cout << "\n--- Screen This Way ---" << endl;
        for (int i = 0; i < ROWS; ++i) {
            cout << "Row " << i + 1 << ":\t";
            Seat* temp = row_heads[i];
            while (temp != nullptr) {
                if (temp->isBooked) {
                    cout << "[B] ";
                } else {
                    cout << "[" << temp->id << "] ";
                }
                temp = temp->next;
            }
```

```
            cout << endl;
        }
        cout << "------------------------\n";
    }

    void bookSeat(int row, int seat_id) {
        if (row < 1 || row > ROWS || seat_id < 1 || seat_id > SEATS_PER_ROW) {
            cout << "Invalid seat selection." << endl;
            return;
        }
        Seat* temp = row_heads[row - 1];
        while (temp != nullptr && temp->id != seat_id) {
            temp = temp->next;
        }
        if (temp != nullptr) {
            if (temp->isBooked) {
                cout << "Seat " << row << "-" << seat_id << " is already booked." << endl;
            } else {
                temp->isBooked = true;
                cout << "Seat " << row << "-" << seat_id << " booked successfully." << endl;
            }
        }
    }

    void cancelBooking(int row, int seat_id) {
        if (row < 1 || row > ROWS || seat_id < 1 || seat_id > SEATS_PER_ROW) {
            cout << "Invalid seat selection." << endl;
            return;
        }
        Seat* temp = row_heads[row - 1];
        while (temp != nullptr && temp->id != seat_id) {
            temp = temp->next;
        }
        if (temp != nullptr) {
            if (!temp->isBooked) {
                cout << "Seat " << row << "-" << seat_id << " is not booked." << endl;
            } else {
                temp->isBooked = false;
                cout << "Booking for seat " << row << "-" << seat_id << " cancelled." << endl;
            }
        }
    }
};

int main() {
    Cinemax theater;
    theater.displaySeats();

    cout << "\nBooking seat 5-6..." << endl;
    theater.bookSeat(5, 6);

    cout << "Booking seat 1-3 again..." << endl;
    theater.bookSeat(1, 3);
```

```
    cout << "Cancelling seat 1-4..." << endl;
    theater.cancelBooking(1, 4);

    theater.displaySeats();

    return 0;
}
```

Output:

```
Seat 1-3 booked successfully.
Seat 1-4 booked successfully.
Seat 5-5 booked successfully.

--- Screen This Way ---
Row 1: [1] [2] [B] [B] [5] [6] [7]
Row 2: [1] [2] [3] [4] [5] [6] [7]
Row 3: [1] [2] [3] [4] [5] [6] [7]
Row 4: [1] [2] [3] [4] [5] [6] [7]
Row 5: [1] [2] [3] [4] [B] [6] [7]
Row 6: [1] [2] [3] [4] [5] [6] [7]
Row 7: [1] [2] [3] [4] [5] [6] [7]
Row 8: [1] [2] [3] [4] [5] [6] [7]
Row 9: [1] [2] [3] [4] [5] [6] [7]
Row 10:          [1] [2] [3] [4] [5] [6] [7]
------------------------

Booking seat 5-6...
Seat 5-6 booked successfully.
Booking seat 1-3 again...
Seat 1-3 is already booked.
Cancelling seat 1-4...
Booking for seat 1-4 cancelled.

--- Screen This Way ---
Row 1: [1] [2] [B] [4] [5] [6] [7]
Row 2: [1] [2] [3] [4] [5] [6] [7]
Row 3: [1] [2] [3] [4] [5] [6] [7]
Row 4: [1] [2] [3] [4] [5] [6] [7]
Row 5: [1] [2] [3] [4] [B] [B] [7]
Row 6: [1] [2] [3] [4] [5] [6] [7]
Row 7: [1] [2] [3] [4] [5] [6] [7]
Row 8: [1] [2] [3] [4] [5] [6] [7]
Row 9: [1] [2] [3] [4] [5] [6] [7]
Row 10:          [1] [2] [3] [4] [5] [6] [7]
------------------------
```

6) infix.cpp

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

using namespace std;
bool areBracketsBalanced(string expr) {
    stack<char> s;
    for (char c : expr) {
        if (c == '(' || c == '[' || c == '{') {
            s.push(c);
            continue;
        }
        if (s.empty()) {
            return false;
        }
    char check;
        switch (c) {
        case ')':
            check = s.top();
            s.pop();
            if (check == '{' || check == '[')
                return false;
            break;
        case '}':
            check = s.top();
            s.pop();
            if (check == '(' || check == '[')
                return false;
            break;
        case ']':
            check = s.top();
            s.pop();
            if (check == '(' || check == '{')
                return false;
            break;
        }
    }
    return (s.empty());
}

int main() {
    string expr1 = "{a + [b * (c - d)]}";
    if (areBracketsBalanced(expr1))
        cout << "Expression '" << expr1 << "' is well-parenthesized." << endl;
    else
        cout << "Expression '" << expr1 << "' is NOT well-parenthesized." << endl;

    string expr2 = "{a + [b * (c - d]})";
     if (areBracketsBalanced(expr2))
        cout << "Expression '" << expr2 << "' is well-parenthesized." << endl;
    else
        cout << "Expression '" << expr2 << "' is NOT well-parenthesized." << endl;
```

```
    return 0;
}
```

Output:

Expression '{a + [b * (c - d)]}' is well-parenthesized.
Expression '{a + [b * (c - d]})' is NOT well-parenthesized.

7) Pizzaparlour.cpp

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

using namespace std;
class PizzaParlor {
private:
   queue<string> orders;
   const int MAX_ORDERS = 5;
public:
   void addOrder(const string& order) {
      if (orders.size() < MAX_ORDERS) {
         orders.push(order);
         cout << "Order placed: " << order << endl;
      } else {
         cout << "Sorry, the queue is full. Cannot accept more orders." << endl;
      }
   }
  void serveOrder() {
      if (!orders.empty()) {
         cout << "Serving order: " << orders.front() << endl;
         orders.pop();
      } else {
         cout << "No orders to serve." << endl;
      }
   }
 void showOrders() {
      if (orders.empty()) {
         cout << "\nThe order queue is empty." << endl;
         return;
      }
      cout << "\nCurrent Orders in Queue:" << endl;
      queue<string> temp = orders;
      int i = 1;
      while (!temp.empty()) {
         cout << i++ << ". " << temp.front() << endl;
         temp.pop();
      }
   }
};
int main() {
   PizzaParlor parlor;
   parlor.addOrder("Margherita Pizza");
   parlor.addOrder("Pepperoni Pizza");
   parlor.addOrder("Veggie Supreme Pizza");
   parlor.showOrders();
   cout << "\n--- Serving Orders ---" << endl;
   parlor.serveOrder();
   parlor.serveOrder();
   parlor.showOrders();
   cout << "\n--- Adding More Orders ---" << endl;
   parlor.addOrder("Hawaiian Pizza");
```

```
    parlor.addOrder("BBQ Chicken Pizza");
    parlor.addOrder("Mushroom Pizza");
    parlor.addOrder("Extra Cheese Pizza"); //This should fail

    parlor.showOrders();

    return 0;
}
```

Output:
Order placed: Margherita Pizza
Order placed: Pepperoni Pizza
Order placed: Veggie Supreme Pizza

Current Orders in Queue:
1. Margherita Pizza
2. Pepperoni Pizza
3. Veggie Supreme Pizza

--- Serving Orders ---
Serving order: Margherita Pizza
Serving order: Pepperoni Pizza

Current Orders in Queue:
1. Veggie Supreme Pizza

--- Adding More Orders ---
Order placed: Hawaiian Pizza
Order placed: BBQ Chicken Pizza
Order placed: Mushroom Pizza
Order placed: Extra Cheese Pizza

Current Orders in Queue:
1. Veggie Supreme Pizza
2. Hawaiian Pizza
3. BBQ Chicken Pizza
4. Mushroom Pizza
5. Extra Cheese Pizza

8) BinaryTree.cpp

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

using namespace std;

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

class BST {
private:
    Node* root;

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

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

    int longestPath(Node* node) {
        if (node == nullptr) return 0;
        return 1 + max(longestPath(node->left), longestPath(node->right));
    }

    Node* findMin(Node* node) {
        if (node == nullptr || node->left == nullptr) return node;
        return findMin(node->left);
    }

    void swapPointers(Node* node) {
        if (node == nullptr) return;
        swap(node->left, node->right);
        swapPointers(node->left);
        swapPointers(node->right);
    }

    bool search(Node* node, int data) {
        if (node == nullptr) return false;
        if (node->data == data) return true;
        if (data < node->data) return search(node->left, data);
        else return search(node->right, data);
    }
```

```
public:
    BST() : root(nullptr) {}

    void insert(int data) { root = insert(root, data); }
    void display() { inorder(root); cout << endl; }
    int findLongestPath() { return longestPath(root); }
    int findMinValue() {
        Node* minNode = findMin(root);
        return (minNode != nullptr) ? minNode->data : -1; // -1 for empty tree
    }
    void swapAllPointers() { swapPointers(root); }
    bool searchValue(int data) { return search(root, data); }
};

int main() {
    BST tree;
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    cout << "Inserting values: 50, 30, 70, 20, 40, 60, 80" << endl;
    for (int val : values) {
        tree.insert(val);
    }

    cout << "Inorder traversal of original tree: ";
    tree.display();

    cout << "Number of nodes in longest path from root: " << tree.findLongestPath() << endl;

    cout << "Minimum data value in the tree: " << tree.findMinValue() << endl;

    int key_to_search = 60;
    cout << "Searching for value " << key_to_search << ": "
         << (tree.searchValue(key_to_search) ? "Found" : "Not Found") << endl;

    cout << "\nSwapping left and right pointers at every node..." << endl;
    tree.swapAllPointers();

    cout << "Inorder traversal of swapped tree: ";
    tree.display();

    return 0;
}
```

Output:
Inserting values: 50, 30, 70, 20, 40, 60, 80
Inorder traversal of original tree: 20 30 40 50 60 70 80
Number of nodes in longest path from root: 3
Minimum data value in the tree: 20
Searching for value 60: Found

Swapping left and right pointers at every node...
Inorder traversal of swapped tree: 80 70 60 50 40 30 20

9) Graphs.cpp

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

using namespace std;

struct DictNode {
    string keyword;
    string meaning;
    DictNode *left, *right;
    DictNode(string key, string mean) : keyword(key), meaning(mean), left(nullptr), right(nullptr) {}
};

class Dictionary {
private:
    DictNode* root;
    int comparisons;

    DictNode* insert(DictNode* node, string key, string mean) {
        if (node == nullptr) return new DictNode(key, mean);
        if (key < node->keyword) node->left = insert(node->left, key, mean);
        else if (key > node->keyword) node->right = insert(node->right, key, mean);
        else node->meaning = mean; // Update meaning if key exists
        return node;
    }

    void displayAscending(DictNode* node) {
        if (node == nullptr) return;
        displayAscending(node->left);
        cout << node->keyword << ": " << node->meaning << endl;
        displayAscending(node->right);
    }

    int findMaxComparisons(DictNode* node, string key) {
        if (node == nullptr) return comparisons;
        comparisons++;
        if (key == node->keyword) return comparisons;
        if (key < node->keyword) return findMaxComparisons(node->left, key);
        else return findMaxComparisons(node->right, key);
    }

    DictNode* findMin(DictNode* node) {
        while (node && node->left != nullptr) node = node->left;
        return node;
    }

    DictNode* deleteNode(DictNode* node, string key) {
        if (node == nullptr) return node;
        if (key < node->keyword) node->left = deleteNode(node->left, key);
        else if (key > node->keyword) node->right = deleteNode(node->right, key);
        else {
            if (node->left == nullptr) {
```

```
            DictNode* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            DictNode* temp = node->left;
            delete node;
            return temp;
        }
        DictNode* temp = findMin(node->right);
        node->keyword = temp->keyword;
        node->meaning = temp->meaning;
        node->right = deleteNode(node->right, temp->keyword);
    }
    return node;
}


public:
    Dictionary() : root(nullptr) {}

    void addKeyword(string key, string mean) {
        root = insert(root, key, mean);
        cout << "Keyword '" << key << "' added/updated." << endl;
    }

    void deleteKeyword(string key) {
        root = deleteNode(root, key);
        cout << "Keyword '" << key << "' deleted." << endl;
    }

    void display() {
        if (root == nullptr) {
            cout << "Dictionary is empty." << endl;
            return;
        }
        cout << "\n--- Dictionary (A-Z) ---" << endl;
        displayAscending(root);
        cout << "-------------------------" << endl;
    }

    int getMaxComparisons(string key) {
        comparisons = 0;
        return findMaxComparisons(root, key);
    }
};

int main() {
    Dictionary dict;
    dict.addKeyword("Algorithm", "A process or set of rules to be followed.");
    dict.addKeyword("Data Structure", "A particular way of organizing data in a computer.");
    dict.addKeyword("Binary", "Relating to, using, or expressed in a system of numerical notation that
has 2 as its base.");
```

```
    dict.display();

    cout << "\nUpdating 'Algorithm'..." << endl;
    dict.addKeyword("Algorithm", "A step-by-step procedure for calculations.");
    dict.display();

    cout << "\nDeleting 'Binary'..." << endl;
    dict.deleteKeyword("Binary");
    dict.display();

    string key_to_find = "Data Structure";
    cout << "\nNumber of comparisons to find '" << key_to_find << "': "
         << dict.getMaxComparisons(key_to_find) << endl;

    return 0;
}
```

Output:

```
Keyword 'Algorithm' added/updated.
Keyword 'Data Structure' added/updated.
Keyword 'Binary' added/updated.

--- Dictionary (A-Z) ---
Algorithm: A process or set of rules to be followed.
Binary: Relating to, using, or expressed in a system of numerical notation that has 2 as its base.
Data Structure: A particular way of organizing data in a computer.
------------------------

Updating 'Algorithm'...
Keyword 'Algorithm' added/updated.

--- Dictionary (A-Z) ---
Algorithm: A step-by-step procedure for calculations.
Binary: Relating to, using, or expressed in a system of numerical notation that has 2 as its base.
Data Structure: A particular way of organizing data in a computer.
------------------------

Deleting 'Binary'...
Keyword 'Binary' deleted.

--- Dictionary (A-Z) ---
Algorithm: A step-by-step procedure for calculations.
Data Structure: A particular way of organizing data in a computer.
------------------------

Number of comparisons to find 'Data Structure': 2
```