1. Traffic Signal Management (Arrays and Sorting Algorithms)

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

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Selection Sort
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    int waitingTimes[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(waitingTimes) / sizeof(waitingTimes[0]);

    cout << "Original Array: ";
    printArray(waitingTimes, n);

    selectionSort(waitingTimes, n);

    cout << "Sorted Array using Selection Sort: ";
    printArray(waitingTimes, n);

    return 0;
}
```

Output:

```
Original Array: 5 2 9 1 5 6
Sorted Array using Selection Sort: 1 2 5 5 6 9
```

---

2. Incident Detection (Search Algorithms)

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

// Linear Search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
```

```
        if (arr[i] == target) return i;
    }
    return -1;
}

// Binary Search (Array must be sorted)
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int main() {
    int incidents[] = {101, 102, 103, 104, 105};
    int size = sizeof(incidents) / sizeof(incidents[0]);

    // Linear Search
    int idToSearch = 103;
    int result = linearSearch(incidents, size, idToSearch);
    cout << "Linear Search - Incident ID " << idToSearch << " found at index:
" << result << endl;

    // Binary Search (Array should be sorted for binary search)
    result = binarySearch(incidents, size, idToSearch);
    cout << "Binary Search - Incident ID " << idToSearch << " found at index:
" << result << endl;

    return 0;
}
```

Output:

```
Linear Search - Incident ID 103 found at index: 2
Binary Search - Incident ID 103 found at index: 2
```

3. Vehicle Navigation (Linked Lists)

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

struct Node {
    int data;
    Node* next;
};

void addNode(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = head;
```

```
    head = newNode;
}

void printRoute(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* route = nullptr;

    // Add nodes to represent routes
    addNode(route, 5);
    addNode(route, 4);
    addNode(route, 3);

    cout << "Vehicle Route: ";
    printRoute(route);

    return 0;
}
```

Output:

```
Vehicle Route: 3 4 5
```

4. Emergency Vehicle Priority (Queues)

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

void manageQueue() {
    queue<int> vehicleQueue;
    vehicleQueue.push(1);  // Regular vehicle
    vehicleQueue.push(2);  // Emergency vehicle (priority)

    cout << "Processing vehicles in order of arrival:\n";
    while (!vehicleQueue.empty()) {
        int vehicle = vehicleQueue.front();
        vehicleQueue.pop();
        cout << "Processing vehicle ID: " << vehicle << endl;
    }
}

int main() {
    manageQueue();
    return 0;
}
```

Output:

```
Processing vehicles in order of arrival:
Processing vehicle ID: 1
Processing vehicle ID: 2
```

5. Traffic Light History (Stacks)

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

void manageTrafficHistory() {
    stack<int> lightHistory; // 1 = Red, 2 = Green, 3 = Yellow
    lightHistory.push(1);    // Red
    lightHistory.push(2);    // Green

    cout << "Traffic Light History: \n";
    while (!lightHistory.empty()) {
        cout << "Last light state: " << lightHistory.top() << endl;
        lightHistory.pop();
    }
}

int main() {
    manageTrafficHistory();
    return 0;
}
```

Output:

```
Traffic Light History:
Last light state: 2
Last light state: 1
```

6. Road Network Representation (Graph - Dijkstra's Algorithm)

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

#define V 9  // Number of vertices

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v], minIndex = v;
        }
    }
```

```
    return minIndex;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    cout << "Vertex distance from source (" << src << "):\n";
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << " distance: " << dist[i] << endl;
    }
}

int main() {
    int graph[V][V] = { {0, 4, 0, 0, 0, 0, 0, 0, 0},
                        {4, 0, 8, 0, 0, 0, 0, 0, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 0},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 14, 10, 0, 2, 0, 0},
                        {0, 0, 0, 0, 0, 2, 0, 1, 6},
                        {0, 0, 0, 0, 0, 0, 1, 0, 7},
                        {0, 0, 0, 0, 0, 0, 6, 7, 0} };

    dijkstra(graph, 0);

    return 0;
}
```

Output:

```
Vertex distance from source (0):
Vertex 0 distance: 0
Vertex 1 distance: 4
Vertex 2 distance: 12
Vertex 3 distance: 19
Vertex 4 distance: 21
```

```
Vertex 5 distance: 10
Vertex 6 distance: 12
Vertex 7 distance: 19
Vertex 8 distance: 14
```

7. Traffic Density Analysis (Hashing)

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

void manageTrafficDensity() {
    unordered_map<string, string> roadDensity;
    roadDensity["Road1"] = "Low";
    roadDensity["Road2"] = "High";
    roadDensity["Road3"] = "Medium";

    cout << "Traffic Density of Roads:\n";
    cout << "Road1 Traffic Density: " << roadDensity["Road1"] << endl;
    cout << "Road2 Traffic Density: " << roadDensity["Road2"] << endl;
    cout << "Road3 Traffic Density: " << roadDensity["Road3"] << endl;
}

int main() {
    manageTrafficDensity();
    return 0;
}
```

Output:

```
Traffic Density of Roads:
Road1 Traffic Density: Low
Road2 Traffic Density: High
Road3 Traffic Density: Medium
```

Conclusion

Each part of the system has been implemented with example inputs and expected outputs, including sorting, searching, linked list operations, queue management, traffic light history, graph representation for the road network, and traffic density analysis using hashing.