FLIGHTS SYSTEM
DOCUMENTATION

2024

Prepared by :
MOHAMMAD FAWAD (FA22-BCS-046)
AMAN EMMANUEL (FA22-BCS-012)

Presented to :
DR. INAYAT-UR-REHMAN

Table of Contents

Flight Dataset Program Documentation

Introduction:

The Flight Dataset program is designed to manage and analyze information about flights using a combination of a doubly linked list for individual flight details and a graph implemented with an adjacency matrix for broader network analysis. The dataset is populated by reading data from a CSV file, and the program offers a menu-driven interface to perform various operations on the dataset.

Data Structures:

1. Linked List
2. Graphs
3. Stack
4. Queue

Algorithms:

1. Dijkstra's Algorithm
2. Prim's Algorithm
3. Hashing Algorithm (Linear Probing)

Doubly Linked List

The doubly linked list serves as the primary structure for storing detailed information about individual flights. Each node in the list represents a flight and contains the following data members:

- **Year:** The year of the flight.
- **Month:** The month of the flight.
- **Day:** The day of the flight.
- **Departure Time:** The time of departure.
- **Departure Delay:** The delay in departure.
- **Arrival Time:** The time of arrival.
- **Arrival Delay:** The delay in arrival.
- **Carrier:** The airline carrier.
- **Tail Number:** The tail number of the aircraft.
- **Flight Number:** The unique identifier for the flight.

- **Origin:** The departure location.
- **Destination:** The arrival location.
- **Airtime:** The duration of the flight.
- **Distance:** The distance covered by the flight.
- **Hours:** The hours component of the flight duration.
- **Minutes:** The minutes component of the flight duration.
- **Previous Pointer:** Pointer to the previous node in the list.
- **Next Pointer:** Pointer to the next node in the list.

Structure of Linked List

```
struct FlightNode
{
    int year;
    int month;
    int day;
    int deptTime;
    int deptdelay;
    int arrTime;
    int arrdelay;
    string carrier;
    string tailnum;
    int flightNo;
    string origin;
    string destination;
    int airTime;
    int distance;
    int hours;
    int minutes;
    FlightNode *next = NULL;
    FlightNode *prev = NULL;
};

FlightNode *first = NULL;
FlightNode *last = NULL;
```

Graph Class:

The **Graph** class is designed to represent and manage a graph structure, with functions encapsulating various operations related to graph manipulation and traversal. In the constructor, the class initializes a graph with a default number of vertices (1000), creating an adjacency matrix to store edge information, as well as arrays for vertex names and indices. The hash function converts string-based vertex names into unique numerical indices. The **getVertexIndex** function retrieves the index of a vertex based on its name, essential for operations like adding edges. The **addVertex** function adds a new vertex to the graph while ensuring its uniqueness, and **addEdge** connects two vertices with a weighted edge. The class supports Depth-First Search (DFS) and Breadth-First Search (BFS) traversal algorithms through the **DFS** and **BFS** functions, respectively. Additionally, it implements Dijkstra's algorithm (**dijkstra**) for finding the shortest paths from a given vertex and Prim's algorithm (**Prims**) for constructing the Minimum Spanning Tree (MST) of the graph. Overall, this **Graph** class provides a comprehensive set of functionalities for creating, modifying, and analyzing graph structures.

Graph Constructor:

```
Graph()
{
    vertices = 1000;
    adjacencyMatrix = new int *[vertices];
    for (int i = 0; i < vertices; i++)
    {
        adjacencyMatrix[i] = new int[vertices];
    }

    vertexNames = new string[vertices];
    vertexIndices = new int[vertices];

    for (int i = 0; i < vertices; i++)
    {
        vertexIndices[i] = -1; // Initialize indices to -1 (invalid index)
        for (int j = 0; j < vertices; j++)
        {
            adjacencyMatrix[i][j] = 0;
        }
    }
}
```

The **Graph** class constructor initializes a graph object by allocating memory for its adjacency matrix, vertex names, and vertex indices. The class assumes a default number of vertices, set to 1000, and dynamically allocates a two-dimensional array (**adjacencyMatrix**) to represent the edges between vertices. It further creates arrays to store the names of vertices (**vertexNames**) and their corresponding indices (**vertexIndices**). During initialization, the indices are set to -1, indicating that no vertices have been added yet. Additionally, the adjacency matrix is initialized with zero values, signifying no edges between vertices initially. Overall, this constructor provides the foundational structure for a graph, preparing it for the addition of vertices and edges as the graph evolves.

Hash Function:

```
int HashFunctiom(string key)
{
    int sum = 0;
    for (int i = 0; i < key.length(); i++)
    {
        char ch = key[i];
        sum += ch;
    }
    return sum % vertices;
}
```

The **HashFunction** converts a string key into an integer hash value by summing the ASCII values of its characters and taking the result modulo the number of vertices in the graph (**vertices**). This ensures a valid index for storing or retrieving data associated with the key in the graph's data structures.

Get Vertex Index Function:

```
int getVertexIndex(string vertex)
{
    int hash = HashFunctiom(vertex);
    int index = hash;

    while (vertexIndices[index] != -1)
    {
        // If Found
        if (vertexNames[index] == vertex)
        {
            return index;
        }
        index = (index + 1) % vertices;
    }

    // If Not found
    return -1;
}
```

The **getVertexIndex** function uses a hashing mechanism to find the index of a vertex in the graph. It first calculates the hash value using the **HashFunction** for the given vertex. It then checks if the calculated index is occupied; if it is, it iterates to the next available index using linear probing until it finds the correct index for the given vertex. If the vertex is found, the corresponding index is returned; otherwise, it returns -1 indicating that the vertex is not present in the graph.

Add Vertex Function:

```
void addVertex(string vertex)
{
    int check = getVertexIndex(vertex);
    if (check == -1)
    {
        int hash = HashFunctiom(vertex);
        int index = hash;

        // If Collision Occurs
        while (vertexIndices[index] != -1)
        {
            index = (index + 1) % vertices;
        }

        vertexNames[index] = vertex;
        vertexIndices[index] = index;
    }
}
```

The **addVertex** function adds a new vertex to the graph. It first checks whether the vertex already exists by calling the **getVertexIndex** function. If the vertex is not found (index is -1), it calculates the hash value for the vertex using the **HashFunction**. In case of a collision, it uses linear probing to find the next available index. Once the appropriate index is determined, the vertex name and index are added to their respective arrays in the graph.

Add Edge Function:

```
void addEdge(string start, string end, int cost)
{
    int startIndex = getVertexIndex(start);
    int endIndex = getVertexIndex(end);

    if (startIndex != -1 && endIndex != -1)
    {
        adjacencyMatrix[startIndex][endIndex] = cost;
        adjacencyMatrix[endIndex][startIndex] = cost;
    }
}
```

The **addEdge** function adds a weighted edge between two vertices in an undirected graph. It first retrieves the indices of the start and end vertices using the **getVertexIndex** function. If both indices are valid, it updates the adjacency matrix to represent the connection between the two vertices with the specified cost. The graph is undirected, so the function updates both **adjacencyMatrix[startIndex][endIndex]** and **adjacencyMatrix[endIndex][startIndex]** to reflect the connection between the vertices.

Depth First Search Function(Queue Implementation):

```
void DFS(string given_start)
{
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    stack<int> q;
    int visited[vertices] = {0};
    int index;

    visited[startIndex] = 1;
    cout << endl
         << "BFS Traversal: ";
    q.push(startIndex);

    while (!q.empty())
    {
        index = q.top();
        cout << vertexNames[index] << " ->";
        q.pop();
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[index][j] != 0 && visited[j] == 0)
            {
                q.push(j);
                visited[j] = 1;
            }
        }
    }
    cout << endl;
}
```

The **DFS** (Depth First Search) function traverses a graph starting from a given vertex using a stack. It begins by obtaining the index of the starting vertex using the **getVertexIndex** function. If the starting vertex is invalid (index is -1), it prints an error message and exits. Otherwise, it initializes a stack, a visited array, and starts the DFS traversal.The function iteratively explores vertices in a depth-first manner. It pops a vertex from the stack, prints its name, and marks it as visited. Then, for each adjacent vertex that is not visited, it pushes that vertex onto the stack and marks it as visited.The traversal continues until the stack is empty. The output displays the DFS traversal sequence of vertices in the graph.

Breath First Search Function(Stack Implementation):

```
void BFS(string given_start)
{
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    queue<int> q;
    int visited[vertices] = {0};
    int index;

    visited[startIndex] = 1;
    cout << endl
         << "BFS Traversal: ";
    q.push(startIndex);

    while (!q.empty())
    {
        index = q.front();
        cout << vertexNames[index] << " -> ";
        q.pop();
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[index][j] != 0 && visited[j] == 0)
            {
                q.push(j);
                visited[j] = 1;
            }
        }
    }
    cout << endl;
}
```

The **BFS** (Breadth First Search) function traverses a graph starting from a given vertex using a queue. It begins by obtaining the index of the starting vertex using the **getVertexIndex** function. If the starting vertex is invalid (index is -1), it prints an error message and exits. Otherwise, it initializes a queue, a visited array, and starts the BFS traversal.The function iteratively explores vertices in a breadth-first manner. It dequeues a vertex, prints its name, and marks it as visited. Then, for each adjacent vertex that is not visited, it enqueues that vertex and marks it as visited.The traversal continues until the queue is empty. The output displays the BFS traversal sequence of vertices in the graph.

Dijkstra Algorithm:

```
void DFS(string given_start)
{
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    stack<int> q;
    int visited[vertices] = {0};
    int index;

    visited[startIndex] = 1;
    cout << endl
         << "BFS Traversal: ";
    q.push(startIndex);

    while (!q.empty())
    {
        index = q.top();
        cout << vertexNames[index] << " ->";
        q.pop();
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[index][j] != 0 && visited[j] == 0)
            {
                q.push(j);
                visited[j] = 1;
            }
        }
    }
    cout << endl;
}
```

The **dijkstra** function in the provided C++ code implements Dijkstra's algorithm for finding the shortest paths from a given starting vertex to all other vertices in a weighted graph. Dijkstra's algorithm maintains an array of distances (**dist**) representing the minimum distance from the starting vertex to each vertex in the graph. It also uses a boolean array (**visited**) to keep track of visited vertices.The algorithm starts by initializing the distance array with infinity (**INT_MAX**) and the starting vertex's distance as 0. It then iterates through the vertices, selecting the vertex with the minimum distance that hasn't been visited. For the selected vertex, it updates the distances to its neighbors if a shorter path is found. This process continues until all vertices are visited or there are no more vertices to visit.The **minDistance** function helps find the vertex with the minimum distance among the unvisited vertices. The **printSolution** function outputs the final distances from the starting vertex to all other vertices.

Prims Algorithm:

```cpp
void Prims(string given_start)
{
    int parent[vertices] = {0};
    int dist[vertices];
    bool visted[vertices];

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

    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    dist[startIndex] = 0;

    for (int count = 0; count < vertices - 1; count++)
    {
        int u = minDistance(dist, visted);
        visted[u] = true;
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[u][j] != 0 && visted[j] == false && adjacencyMatrix[u][j] < dist[j])
            {

                parent[j] = u;
                dist[j] = adjacencyMatrix[u][j];
            }
        }
    }

    printSolution1(parent, startIndex);
}
```

The **Prims** function in the provided C++ code implements Prim's algorithm for finding the Minimum Spanning Tree (MST) of a connected, undirected graph with weighted edges. Prim's algorithm grows a tree from a starting vertex by adding the minimum-weight edge at each step.The algorithm uses an array **parent** to keep track of the parent of each vertex in the MST, **dist** to store the minimum weight of the edge connecting each vertex to the MST, and **visited** to mark vertices that have already been included in the MST.The algorithm starts by initializing the **dist** array with infinity (**INT_MAX**) and the starting vertex's distance as 0. It then iteratively selects the vertex with the minimum distance to the MST, adds it to the MST, and updates the distances to its neighbors if a shorter edge is found.The **printSolution1** function outputs the edges of the MST along with their weights. It excludes edges where the parent is 0 or the start vertex to avoid displaying unnecessary information.

Main Program:

The program reads flight data from a CSV file named "FlightsDataset.csv" using an ifstream named **fin**. It then processes each line of the file, splitting it into individual data elements using a comma as a delimiter. The data elements are stored in a vector called **lineData**. The program extracts various attributes from **lineData** using stringstreams and converts them into their respective data types. Specifically, it converts strings to integers for attributes like year, month, day, departure time, departure delay, arrival time, arrival delay, flight number, air time, distance, hours, and minutes.

Following this data extraction, the program adds the flight data to a linked list using the **addFlightNode** function. The **addFlightNode** function takes the flightgraph and the extracted attributes as parameters, creating a node in the graph for each flight.

The process iterates until the end of the file is reached. The variable **lines** keeps track of the number of lines read from the file.

This approach efficiently parses the CSV data, converts it to the appropriate data types, and integrates it into the flightgraph data structure, likely representing a graph of flight connections or related information. The code adheres to a structured and systematic method of handling the dataset.

```
int main()
{
    Graph flightgraph;
    ifstream fin;
    fin.open("FlightsDataset.csv");
    string line, word;
    vector<string> lineData;
    int lines = 0;

    getline(fin, line);

    while (!fin.eof())
    {
        getline(fin, line);
        stringstream s(line);
        while (getline(s, word, ','))
        {
            lineData.push_back(word);
        }

        stringstream s0(lineData[0]);
        int year;
        s0 >> year;

        stringstream s1(lineData[1]);
        int month;
        s1 >> month;

        stringstream s2(lineData[2]);
        int day;
        s2 >> day;

        stringstream s3(lineData[3]);
        int deptTime;
        s3 >> deptTime;

        stringstream s4(lineData[4]);
        int deptdelay;
        s4 >> deptdelay;

        stringstream s5(lineData[5]);
        int arrTime;
        s5 >> arrTime;
```

```cpp
        stringstream s6(lineData[6]);
        int arrdelay;
        s6 >> arrdelay;

        stringstream s9(lineData[9]);
        int flightNo;
        s9 >> flightNo;

        stringstream s12(lineData[12]);
        int airTime;
        s12 >> airTime;

        stringstream s13(lineData[13]);
        int distance;
        s13 >> distance;

        stringstream s14(lineData[14]);
        int hours;
        s14 >> hours;

        stringstream s15(lineData[15]);
        int minutes;
        s15 >> minutes;

        // adding data to linked list
        addFlightNode(flightgraph, year, month, day, deptTime, deptdelay, arrTime, arrdelay, lineData[7], lineData[8], flightNo, lineData[10], lineData[11], airTime, distance, hours, minutes);

        lineData.clear();
        lines++;
    }
```

Menu Options:

The menu options provide users with a diverse set of functionalities to explore and analyze the flight dataset. Each option corresponds to a specific operation, facilitating easy navigation and interaction with the dataset.

```cpp
void displaymenu()
{
    cout << "\n1. Insert A New Flight in List" << endl;
    cout << "2. Delete a flight from the list with its Carrier,TailNumber and FlightNo" << endl;
    cout << "3. Print information about all flights" << endl;
    cout << "4. Print Information About Flights of A Year" << endl;
    cout << "5. Print Information About Flights of A Year with its month" << endl;
    cout << "6. Print Information about flights of same year, month, and day" << endl;
    cout << "7. Print Flights by Carrier" << endl;
    cout << "8. Print Flights Of Same Origin" << endl;
    cout << "9. Print Flights Of Same Destination" << endl;
    cout << "10. Print Flights by Tail Number" << endl;
    cout << "11. Print Short Time Flights with flight time less than given time" << endl;
    cout << "12. Print Short Time Flights with flight time greater than given time" << endl;
    cout << "13. Print all Flights Information of given distance" << endl;
    cout << "14. Print all Flights Information with distance lesser than given range" << endl;
    cout << "15. Print all Flights Information with distance greater than given range" << endl;
    cout << "16. Print flights by origin and departure time" << endl;
    cout << "17. Print flights by destination and air time" << endl;
    cout << "18. Print flights by Specific destination and arrival time" << endl;
    cout << "19. Print Highest Depture Dealyed Flight" << endl;
    cout << "20. Print Highest Arrival Dealyed Flight" << endl;
    cout << "21. Print All Flights with your given origin and dept Flying time" << endl;
    cout << "22. Print The Most Longest Distance Flight" << endl;
    cout << "23. Print The Most Shortest Distance Flight" << endl;
    cout << "24. Delete All The Flights of Given Carrier" << endl;
    cout << "25. Delete All The Flights of Given TailNum" << endl;
    cout << "26. Delete All The Flights of Given Origin" << endl;
    cout << "27. Delete All The Flights of Given Destination" << endl;
    cout << "28. Print Average of All Flights Distance " << endl;
    cout << "29.Print The Fastest Speed Flight Infomration" << endl;
    cout << "30.Print The Slowest Speed Flight Infomration" << endl;
    cout << "31.Breadth First Traversal of Graph" << endl;
    cout << "32.Depth First Traversal of Graph" << endl;
    cout << "33.Apply Dijkstra Thoeorm" << endl;
    cout << "34.Apply Prims's Thoeorm" << endl;
    cout << "0. Exit" << endl;
}
```

Option 1: Insert A New Flight in List:

This option allows the user to add a new flight to the linked list.

```
void addFlightNode(int yeaar, int mnth, int dy, int dptTme, int deptdelay, int arrivtme, int arrivedly, string carier,
                   string tailnum, int flight_no, string orgin, string dest, int airtim, int dstanc, int hrs, int mntues)
{
    FlightNode *crnt = new FlightNode;
    crnt->year = yeaar;
    crnt->month = mnth;
    crnt->day = dy;
    crnt->deptTime = dptTme;
    crnt->deptdelay = deptdelay;
    crnt->arrTime = arrivtme;
    crnt->arrdelay = arrivedly;
    crnt->carrier = carier;
    crnt->tailnum = tailnum;
    crnt->flightNo = flight_no;
    crnt->origin = orgin;
    crnt->destination = dest;
    crnt->airTime = airtim;
    crnt->distance = dstanc;
    crnt->hours = hrs;
    crnt->minutes = mntues;

    if (first == NULL)
    {
        first = crnt;
        last = crnt;
        crnt->next = NULL;
    }
    else
    {
        // Insert at End
        crnt->prev = last;
        last->next = crnt;
        last = crnt;
    }
}
```

To create a new node in the doubly linked list, a FlightNode named **crnt** is dynamically allocated, and flight attributes are assigned accordingly. If the list is empty (i.e., **first** is NULL), the new node becomes both the first and last node in the list. In case the list is not empty, the new node is inserted at the end. The **prev** pointer of the new node (**crnt**) is set to point to the current last node, and the **next** pointer of the current last node is set to point to the new node (**crnt**). Finally, the **last** pointer is updated to point to the new last node (**crnt**). This insertion mechanism ensures the proper linking of nodes in a doubly linked list, facilitating efficient traversal in both forward and backward directions.

Option 2: Delete a Flight from the List with Carrier, TailNumber, and FlightNo:

This option allows the user to delete a specific flight from the list using carrier, tail number, and flight number.

```
// Delete A Single Flight by Its Carrier,tialnum and FlightNum
void deleteFlightByDetails(string carrier, string tailNumber, int flightNumber)
{
    FlightNode *current = first;

    while (current != NULL)
    {
        if (current->carrier == carrier && current->tailnum == tailNumber && current->flightNo == flightNumber)
        {
            if (first == last)
            {
                first = last = NULL;
            }
            else if (current == first)
            {
                first = first->next;
                first->prev = NULL;
            }
            else if (current == last)
            {
                last = last->prev;
                last->next = NULL;
            }
            else
            {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }

            delete current;
            cout << "Flight with Carrier " << carrier << ", Tail Number " << tailNumber
                 << ", and Flight Number " << flightNumber << " deleted successfully." << endl;
            return;
        }

        current = current->next;
    }

    cout << "Flight with Carrier " << carrier << ", Tail Number " << tailNumber
         << ", and Flight Number " << flightNumber << " not found." << endl;
}
```

First, the pointer '**current**' is initialized to the beginning of the list. The function then traverses the list, scrutinizing each node until a match is found based on specified criteria such as **carrier**, **tail number**, and **flight number**. Upon locating the target node, the deletion process unfolds: if the node is the sole element, both the '**first**' and '**last**' pointers are set to **NULL**. In the case of the first node, '**first**' is updated to the next node, and the '**prev**' pointer of the new first node is set to **NULL**. For the last node, '**last**' is adjusted to point to the preceding node, and the '**next**' pointer of the new last node is set to **NULL**. If the node is neither the first nor the last, the '**prev**' pointer of the subsequent node and the '**next**' pointer of the prior node are modified to bypass the current node. Finally, the memory occupied by the node to be deleted ('**current**') is deallocated using the '**delete**' operator. The process concludes with a status message indicating the success of the deletion or the absence of a matching node.

Option 3: Print Information About All Flights:

This option prints detailed information about all flights in the dataset.

```cpp
// printing information about all flights
void printAllFlights()
{
    FlightNode *p = first;
    cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
    if (first != NULL)
    {
        while (p != NULL)
        {
            cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                 << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                 << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            p = p->next;
        }
    }
    else
    {
        cout << "Theere is no data present to display" << endl;
    }
}
```

The function initializes a pointer '**p**' to the first node in the doubly linked list and prints a header for flight information. It checks for an empty list, and if not, iterates through, printing flight details such as year, month, day, departure time, etc. If the list is empty, a corresponding message is displayed.

Option 4: Print Information About Flights of a Year:

This option prints information about flights that occurred in a specific year.

```cpp
// All Flights of Same Year
void pirntFlightsByYear(int givenyear)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "Theere is no Elemts Prent in the liist";
    }
    else
    {
        cout << "\n All Flights With Given Year are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
        while (p != NULL)
        {
            if (p->year == givenyear)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function initializes a pointer '**p**' to the first node in the doubly linked list. It checks if the list is empty; if so, it prints a message about the lack of elements. If not, it prints a header for flight information and iterates through the list. For each node, it checks if the year matches the given year, and if so, prints the details of that flight.

Option 5: Print Information About Flights of a Year with Its Month:

This option prints information about flights in a specific year and month.

```cpp
// All Flights of Same Year and Same Moneth
void pirntFlightsOfYearMon(int givenyear, int givenMont)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "Theere is no Elemts Prent in the liist";
    }
    else
    {
        cout << "\n All Flights of Given Year and month are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->year == givenyear && p->month == givenMont)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function initializes a pointer '**p**' to the first node in the doubly linked list. It checks if the list is empty; if so, it prints a message about the lack of elements. If not, it prints a header for flight information and iterates through the list. For each node, it checks if the year and month match the given year and month, and if so, prints the details of that flight.

Option 6: Print Information About Flights of the Same Year, Month, and Day:

This option prints information about flights that occurred on the same year, month, and day.

```cpp
// All Flights of Same Year and Same Moneth and Same Day
void pirntFlightsOfYearMontDay(int givenyear, int givenMont, int givenday)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "Theere is no Elemts Prent in the liist";
    }
    else
    {
        cout << "\n All Flights of Given same Year and month and Day are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->year == givenyear && p->month == givenMont && p->day == givenday)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating the absence of elements. If the list is not empty, the function proceeds to print flights with the given year, month, and day. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the year, month, and day match the provided values. If all conditions are met, the details of that flight are printed.

Option 7: Print Flights by Carrier:

This option prints information about flights operated by a specific carrier.

```cpp
// Print All Flights of An Carrier
void printFlightsByCarrier(string givenCarrier)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There are no elements present in the list";
    }
    else
    {
        cout << "\n All Flights of Given Carrier are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->carrier == givenCarrier)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function initiates by initializing a pointer '**p**' to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating the absence of elements. If the list is not empty, the function proceeds to print flights with the given carrier. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the carrier matches the provided value. If the carrier matches, the details of that flight are printed.

Option 8: Print Flights of Same Origin:

This option prints information about flights with the same origin.

```cpp
void printFlightsByOrigin(string givenOrigin)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There are no elements present in the list";
    }
    else
    {
        cout << "\n All Flights of Given Origin are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->origin == givenOrigin)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating the absence of elements. If the list is not empty, the function proceeds to print flights with the given origin. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the origin matches the provided value. If the origin matches, the details of that flight are printed.

Option 9: Print Flights of Same Destination:

This option prints information about flights with the same destination.

```cpp
void printFlightsByDestination(string givenDestination)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There are no elements present in the list";
    }
    else
    {
        cout << "\n All Flights to Given Destination are: \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->destination == givenDestination)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function starts by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating the absence of elements. If the list is not empty, the function proceeds to print flights with the given destination. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the destination matches the provided value. If the destination matches, the details of that flight are printed.

Option 10: Print Flights by Tail Number:

This option prints information about flights based on a specific tail number.

```cpp
// Function to print ALL flights with a same Tail numbers
void printFlightsByTailNum(string giventailnum)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "All Flight Information for Given Tail Number " << giventailnum << ":" << endl;
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        while (p != NULL)
        {
            if (p->tailnum == giventailnum)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights with the given tail number. A header line is printed to indicate the format of the flight information. Following this, a while loop iterates through the list, and for each node, it checks if the tail number matches the provided value. If the tail number matches, the details of that flight are printed.

Option 11: Print Short Time Flights with Flight Time Less Than Given Time:

This option prints information about flights with a duration less than a given time.

```cpp
void printShortDurationFlights(int given_maxduration)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights with Duration Less Than " << given_maxduration << " minutes:" << endl;
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
        while (p != NULL)
        {
            int totalMinutes = p->hours * 60 + p->minutes;
            if (totalMinutes < given_maxduration)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function starts by initializing a pointer '**p**' to the first node in the doubly linked list. It then checks if the list is empty; if so, it prints a message indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights with a duration less than the given maximum duration. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it calculates the total duration in minutes (sum of hours and minutes) and checks if it is less than the provided maximum duration. If the duration is less than the given maximum, the details of that flight are printed.

Option 12: Print Short Time Flights with Flight Time Greater Than Given Time:

This option prints information about flights with a duration greater than a given time.

```cpp
void printLongDurationFlights(int given_minduration)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights with Duration Greater Than " << given_minduration << " minutes:" << endl;
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
        while (p != NULL)
        {
            int totalMinutes = p->hours * 60 + p->minutes;
            if (totalMinutes > given_minduration)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights with a duration greater than the given minimum duration. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it calculates the total duration in minutes (sum of hours and minutes) and checks if it is greater than the provided minimum duration. If the duration is greater than the given minimum, the details of that flight are printed.

Option 13: Print All Flights Information of Given Distance:

This option prints information about flights with a specific distance.

```
void printFlightbyDistance(int given_dist)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There are no elements present in the list";
    }
    else
    {
        cout << "\n All Flights of Given distance" << given_dist << " : \n";
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;

        while (p != NULL)
        {
            if (p->distance == given_dist)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function starts by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there are no elements to display. If the list is not empty, the function proceeds to print flights with a distance equal to the given distance. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the distance is equal to the provided distance. If the distance matches the given value, the details of that flight are printed.

Option 14: Print All Flights Information with Distance Lesser Than Given Range:

This option prints information about flights with a distance less than a given range.

```
// Print all flights with distnce less than given distnace
void printLessDistanceFlights(int given_disance)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights with Distance Shorter Than " << given_disance << " disatnce:" << endl;
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
        while (p != NULL)
        {
            if (p->distance < given_disance)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights with a distance shorter than the given distance. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the distance is less than the provided distance. If the distance is less than the given value, the details of that flight are printed.

Option 15: Print All Flights Information with Distance Greater Than Given Range:

This option prints information about flights with a distance greater than a given range.

```
void printMoreDistanceFlights(int given_disance)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights with Distance Graeter Than " << given_disance << " disatnce:" << endl;
        cout << "Year , Month , Day, Dept Time ,Dept Delay, Arrival Time ,Arrival Delay , Carrier ,TailNum, Flight No , Origin , Destination , Air Time , Distance, Hours, Minutes" << endl;
        while (p != NULL)
        {
            if (p->distance > given_disance)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                    << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                    << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights with a distance greater than the given distance. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the distance is greater than the provided distance. If the distance is greater than the given value, the details of that flight are printed.

Option 16: Print Flights by Origin and Departure Time:

This option prints information about flights based on origin and departure time.

```
void printFlightsByOriginAndDepartureTime(string give_origin, int give_deptTime)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights from " << give_origin << " with Departure Time at " << give_deptTime << ":" << endl;
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        while (p != NULL)
        {
            if (p->origin == give_origin && p->deptTime == give_deptTime)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                    << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                    << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights originating from the given location with the specified departure time. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the origin matches the given origin and if the departure time matches the given departure time. If both conditions are met, the details of that flight are printed.

Option 17: Print Flights by Destination and Airtime:

This option prints information about flights based on destination and airtime.

```cpp
void printFlightsByDestinationAndAirTime(string given_destination, int airtme)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights to " << given_destination << " with Air Time " << airtme << " minutes:" << endl;
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        while (p != NULL)
        {
            if (p->destination == given_destination && p->airTime == airtme)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function starts by initializing a pointer '**p**' to point to the first node in the doubly linked list. It then checks if the list is empty; if it is, a message is printed indicating that there is nothing to display. If the list is not empty, the function proceeds to print flights destined for the given location with the specified airtime. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the destination matches the given destination and if the airtime matches the given airtime. If both conditions are met, the details of that flight are printed.

Option 18: Print Flights by Specific Destination and Arrival Time:

This option prints information about flights based on a specific destination and arrival time.

```cpp
void printFlightsByDestinationAndArrivalTime(string destination, int arrTime)
{
    FlightNode *p = first;
    if (p == NULL)
    {
        cout << "There is Nothing To Display" << endl;
    }
    else
    {
        cout << "Flights to " << destination << " with Arrival Time at " << arrTime << ":" << endl;
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        while (p != NULL)
        {
            if (p->destination == destination && p->arrTime == arrTime)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function begins by initializing a pointer '**p**' to point to the first node in the doubly linked list. After that, it checks whether the list is empty. If the list is empty (i.e., '**first**' is **NULL**), a message is displayed, indicating that there is nothing to display. However, if the list is not empty, the function proceeds to print flights destined for the given location with the specified arrival time. A header line is printed to indicate the format of the flight information. Subsequently, a while loop iterates through the list, and for each node, it checks if the destination matches the given destination and if the arrival time matches the given arrival time. If both conditions are satisfied, the details of that flight are printed.

Option 19: Print Highest Departure Delayed Flight:

This option prints information about the flight with the highest departure delay.

```
void printHighestDeptDealyFlight()
{
    if (first == NULL)
    {
        cout << "There is No Elemnts in List";
    }
    else
    {
        FlightNode *crnt = first;

        FlightNode *p = first;
        int highest = first->deptdelay;

        while (crnt != NULL)
        {
            if (crnt->deptdelay > highest)
            {
                highest = crnt->deptdelay;
                p = crnt;
            }
            crnt = crnt->next;
        }
```

The function checks for an empty list and, if not empty, finds and prints details of the flight with the highest departure delay. It uses two pointers, '**crnt**' and '**p**', initialized to the first node, with '**p**' tracking the highest delay. After iterating through the list, it prints the details of the flight with the highest departure delay.

Option 20: Print Highest Arrival Delayed Flight:

This option prints information about the flight with the highest arrival delay.

```
void printHighestArrivalDealyFlight()
{
    if (first == NULL)
    {
        cout << "There is No Elemnts in List";
    }
    else
    {
        FlightNode *crnt = first;

        FlightNode *p = first;
        int highest = first->arrdelay;

        while (crnt != NULL)
        {
            if (crnt->arrdelay > highest)
            {
                highest = crnt->arrdelay;
                p = crnt;
            }
            crnt = crnt->next;
        }

        cout << "The Flight with highest arrival delay is: " << endl;
        cout
            << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
            << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
            << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
    }
}
```

The function checks for an empty list and, if not empty, finds and prints details of the flight with the highest arrival delay. It uses two pointers, '**crnt**' and '**p**', initialized to the first node, with '**p**' tracking the highest delay. After iterating through the list, it prints the details of the flight with the highest arrival delay.

Option 21: Print All Flights with Your Given Origin and Departure Flying Time:

This option prints information about all flights with a user-specified origin and departure flying time.

```
void printFlightbyOriginDeptTime(string given_origin, int time)
{
    if (first == NULL)
    {
        cout << "There is nothing to display";
    }
    else
    {
        FlightNode *p = first;
        int departuretime;

        cout << "All Flights of Your Given Time and Origin are: ";
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;

        while (p != NULL)
        {
            departuretime = p->deptTime + p->deptdelay;
            if (departuretime == time && p->origin == given_origin)
            {
                cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
                     << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
                     << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
            }
            p = p->next;
        }
    }
}
```

The function checks for an empty list and, if not empty, iterates through the list to find and print details of flights with a specified departure time and origin. It calculates the actual departure time by considering the departure delay. If a match is found, it prints the details of the matching flights.

Option 22: Print The Most Longest Distance Flight:

This option prints information about the flight with the longest distance.

```
void printLongestDistanceFlight()
{
    if (first == NULL)
    {
        cout << "There is No Elemnts in List";
    }
    else
    {
        FlightNode *crnt = first;

        FlightNode *p = first;
        int highest = first->distance;

        while (crnt != NULL)
        {
            if (crnt->distance > highest)
            {
                highest = crnt->distance;
                p = crnt;
            }
            crnt = crnt->next;
        }

        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
            << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
            << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
    }
}
```

The function checks for an empty list and, if not empty, iterates through the list to find and print details of the flight with the longest distance. It compares the distance of each node with the highest distance encountered so far and updates it accordingly. After the loop, it prints the details of the flight with the longest distance.

Option 23: Print The Shortest Distance Flight:

This option prints information about the flight with the shortest distance.

```cpp
void printShortestDistanceFlight()
{
    if (first == NULL)
    {
        cout << "There is No Elemnts in List";
    }
    else
    {
        FlightNode *crnt = first;

        FlightNode *p = first;
        int highest = first->distance;

        while (crnt != NULL)
        {
            if (crnt->distance < highest)
            {
                highest = crnt->distance;
                p = crnt;
            }
            crnt = crnt->next;
        }

        cout << "The Shoretst Distance Flight Info is: " << endl;
        cout << "Dept Time , Arr Time , Sched Arrival , Depart Arrival , Carrier , Flight No , Origin , Dest , Air Time , Distance , Hours , Minutes" << endl;
        cout << p->year << " , " << p->month << " , " << p->day << " , " << p->deptTime << " , " << p->deptdelay << " , " << p->arrTime << " , " << p->arrdelay << " , "
             << p->carrier << " , " << p->tailnum << " , " << p->flightNo << " , " << p->origin << " , " << p->destination << " , "
             << p->airTime << " , " << p->distance << " , " << p->hours << " , " << p->minutes << endl;
    }
}
```

The function checks for an empty list and, if not empty, iterates through the list to find and print details of the flight with the shortest distance. It compares the distance of each node with the highest distance (initialized to the distance of the first node) to find the shortest distance. After the loop, it prints the details of the flight with the shortest distance.

Option 24: Delete All the Flights of Given Carrier:

This option deletes all flights operated by a specific carrier.

```cpp
void deleteFlightsByCarrier(string carrier)
{
    FlightNode *current = first;
    while (current != NULL)
    {
        if (current->carrier == carrier)
        {
            if (first == last)
            {
                first = last = NULL;
            }
            else if (current == first)
            {
                first = first->next;
                first->prev = NULL;
            }
            else if (current == last)
            {
                last = last->prev;
                last->next = NULL;
            }
            else
            {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }
            delete current;
        }
        current = current->next;
    }
    cout << "All Flights with given carrier Name has been deleted";
}
```

The function first checks for an empty list and prints a corresponding message. If the list is not empty, it iterates through the list using a while loop. For each node, it checks if the carrier's name matches the given carrier. If a match is found, the function deletes the node, considering cases for the first, last, or intermediate nodes. After deletion, it adjusts the pointers of adjacent nodes and prints a message indicating the deletion of all flights with the given carrier's name.

Option 25: Delete All the Flights of Given Tail Number:

This option deletes all flights associated with a specific tail number.

```
void deleteFlightsByTailNumber(string given_tailNumber)
{
    FlightNode *current = first;
    while (current != NULL)
    {
        if (current->tailnum == given_tailNumber)
        {
            if (first == last)
            {
                first = last = NULL;
            }
            else if (current == first)
            {
                first = first->next;
                first->prev = NULL;
            }
            else if (current == last)
            {
                last = last->prev;
                last->next = NULL;
            }
            else
            {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }
            delete current;
        }
        current = current->next;
    }
    cout << "All Flights with given Tailnumber has been deleted";
}
```

The function checks for an empty list and prints a corresponding message. If the list is not empty, it iterates through the list using a while loop. For each node, it checks if the tail number matches the given tail number. If a match is found, the function deletes the node, considering cases for the first, last, or intermediate nodes. After deletion, it adjusts the pointers of adjacent nodes and prints a message indicating the deletion of all flights with the given tail number.

Option 26: Delete All the Flights of Given Origin:
This option deletes all flights with a specific origin.

```cpp
void deleteFlightsByOrigin(string origin)
{
    FlightNode *current = first;
    while (current != NULL)
    {
        if (current->origin == origin)
        {
            if (first == last)
            {
                first = last = NULL;
            }
            else if (current == first)
            {
                first = first->next;
                first->prev = NULL;
            }
            else if (current == last)
            {
                last = last->prev;
                last->next = NULL;
            }
            else
            {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }
            delete current;
        }
        current = current->next;
    }
    cout << "All Flights with given Origin Airport has been deleted";
}
```

The function begins by checking for an empty list, printing a corresponding message if the list is empty. If the list contains elements, it utilizes a while loop to iterate through each node. For every node, the function checks if the origin airport matches the provided origin. Upon finding a match, the node is deleted, with deletion logic accommodating scenarios where the node is the first, last, or an intermediate node in the doubly linked list. After node deletion, the pointers (prev and next) of adjacent nodes are adjusted accordingly. Finally, the function prints a message confirming the deletion of all flights with the given origin airport.

Option 27: Delete All the Flights of Given Destination:
This option deletes all flights with a specific destination.

```cpp
void deleteFlightsByDestination(string given_destination)
{
    FlightNode *current = first;
    while (current != NULL)
    {
        if (current->destination == given_destination)
        {
            if (first == last)
            {
                first = last = NULL;
            }
            else if (current == first)
            {
                first = first->next;
                first->prev = NULL;
            }
            else if (current == last)
            {
                last = last->prev;
                last->next = NULL;
            }
            else
            {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }
            delete current;
        }
        current = current->next;
    }
    cout << "All Flights with given Same Destination Airport has been deleted";
}
```

The function first checks if the list is empty, printing a message if it is. Assuming the list has elements, the function iterates through the list using a while loop. For each node, it checks if the destination airport matches the provided destination. Upon finding a match, the node is deleted, with deletion logic accounting for scenarios where the node is the first, last, or an intermediate node in the doubly linked list. After the deletion, the pointers (prev and next) of adjacent nodes are adjusted accordingly. Finally, the function prints a message confirming the deletion of all flights with the specified destination airport.

Option 28: Print Average of All Flights Distance:

This option calculates and prints the average distance of all flights.

```
void printAverageFlightDistance()
{
    if (first == NULL)
    {
        cout << "ther is nothing to display";
    }
    else
    {

        int totalDistance = 0;
        int numberOfFlights = 0;

        FlightNode *current = first;
        while (current != NULL)
        {
            totalDistance += current->distance;
            numberOfFlights++;
            current = current->next;
        }

        double averageDistance = static_cast<double>(totalDistance) / numberOfFlights;
        cout << "Average Flight Distance: " << averageDistance << " miles" << endl;
    }
}
```

The function begins by checking if the list is empty, printing a message if it is. If the list is not empty, the function initializes variables totalDistance and numberOfFlights to zero. It then employs a while loop to iterate through the list, accumulating the total distance and counting the number of flights. Following the loop, the function calculates the average distance by dividing the totalDistance by the numberOfFlights, storing the result in averageDistance. Finally, the function prints the calculated average distance in miles.

Option 29: Print The Fastest Speed Flight Information:
This option prints information about the flight with the fastest speed.

```cpp
void printFastestAverageSpeedFlight()
{
    if (first == NULL)
    {
        cout << "ther is nothing to display";
    }
    else
    {

        FlightNode *current = first;
        FlightNode *fastestFlight = first;

        double highestSpeed = 0.0;
        double cuurentspped;

        while (current != NULL)
        {
            cuurentspped = static_cast<double>(current->distance) / current->airTime;

            if (cuurentspped > highestSpeed)
            {
                highestSpeed = cuurentspped;
                fastestFlight = current;
            }

            current = current->next;
        }

        cout << "Flight with the fastest average speed: " << fastestFlight->carrier << " "
             << fastestFlight->flightNo << " from " << fastestFlight->origin << " to "
             << fastestFlight->destination << " with an average speed of " << highestSpeed << " miles per minute." << endl;
    }
}
```

The function starts by checking if the list is empty, printing a message if it is. If the list is not empty, the function initializes variables fastestFlight to the first node and highestSpeed to zero. It then uses a while loop to iterate through the list, calculating the average speed for each flight using the formula distance / airTime. If the calculated speed is greater than the current highest speed, the variables fastestFlight and highestSpeed are updated. After the loop, the function prints information about the flight with the fastest average speed, including carrier, flight number, origin, destination, and the calculated average speed.

Option 30: Breadth-First Traversal of Graph:
This option performs a breadth-first traversal of the graph.

```cpp
// BFS Traversal OF Graph

void BFS(string given_start)
{
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    queue<int> q;
    int visited[vertices] = {0};
    int index;

    visited[startIndex] = 1;
    cout << endl
         << "BFS Traversal: ";
    q.push(startIndex);

    while (!q.empty())
    {
        index = q.front();
        cout << vertexNames[index] << " -> ";
        q.pop();
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[index][j] != 0 && visited[j] == 0)
            {
                q.push(j);
                visited[j] = 1;
            }
        }
    }
    cout << endl;
}
```

1. **Function Signature:** The function **BFS** takes the name of a starting vertex (**given_start**) as input and performs a Breadth-First Search (BFS) traversal on the graph.
2. **Error Handling:** It checks the validity of the given starting vertex by using **getVertexIndex(given_start)**. If the index is -1, indicating an invalid vertex, it prints an error message and returns from the function.
3. **Queue Initialization:** The function uses a standard queue (**std::queue**) to keep track of vertices during the BFS traversal.
4. **Visited Array:** To keep track of visited vertices and avoid processing them multiple times, the function uses a boolean array (**visited**). This array is initialized to **false** for all vertices at the beginning.
5. **Traversal:** The BFS traversal starts from the given vertex. It marks the vertex as visited, prints its name, and explores its neighbors. This process continues until the queue is empty. The traversal follows the FIFO (First-In-First-Out) principle.
6. **Graph Representation:** The graph is assumed to be represented using an adjacency matrix (**adjacencyMatrix**). The function uses **vertexNames** to print the names of vertices during traversal.
7. **Output:** The BFS traversal sequence is printed as vertices are visited, providing insight into the order in which vertices are discovered and processed during the BFS traversal.

Option 31: Depth-First Traversal of Graph:

This option performs a depth-first traversal of the graph.

```cpp
// DFS Tarversal Of Graph

void DFS(string given_start)
{
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    stack<int> q;
    int visited[vertices] = {0};
    int index;

    visited[startIndex] = 1;
    cout << endl
         << "BFS Traversal: ";
    q.push(startIndex);

    while (!q.empty())
    {
        index = q.top();
        cout << vertexNames[index] << " ->";
        q.pop();
        for (int j = 0; j < vertices; j++)
        {
            if (adjacencyMatrix[index][j] != 0 && visited[j] == 0)
            {
                q.push(j);
                visited[j] = 1;
            }
        }
    }
    cout << endl;
}
```

1. **Function Signature:** The **DFS** function takes the name of a starting vertex (**given_start**) as input and performs a Depth-First Search (DFS) traversal on the graph.

2. **Error Handling:** The function checks the validity of the given starting vertex by using **getVertexIndex(given_start)**. If the index is -1, indicating an invalid vertex, the function prints an error message and returns.

3. **Stack Initialization:** A standard stack (**std::stack**) is employed to keep track of vertices during the DFS traversal.

4. **Visited Array:** To keep track of visited vertices and avoid redundant processing, the function uses a boolean array (**visited**). This array is initially set to **false** for all vertices.

5. **Traversal:** The DFS traversal begins from the given vertex. It marks the vertex as visited, prints its name, and explores its neighbors using a stack. This process continues until the stack is empty. The traversal follows the LIFO (Last-In-First-Out) principle.

6. **Graph Representation:** The graph is assumed to be represented using an adjacency matrix (**adjacencyMatrix**). The function uses **vertexNames** for printing the names of vertices during traversal.

7. **Output:** The DFS traversal sequence is printed as vertices are visited, revealing the order in which vertices are discovered and processed during the DFS traversal.

Option 32: Apply Dijkstra's Algorithm:

This option applies Dijkstra's algorithm to find the shortest path in the graph.

```cpp
void dijkstra(string given_start)
{
    // First Check Whater City Exist or Not
    int startIndex = getVertexIndex(given_start);
    if (startIndex == -1)
    {
        cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
        return;
    }

    int dist[vertices];
    bool visited[vertices];

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

    dist[startIndex] = 0;

    for (int count = 0; count < vertices - 1; count++)
    {
        int u = minDistance(dist, visited);
        if (u == -1)
            break;

        visited[u] = true;

        for (int v = 0; v < vertices; v++)
        {
            if (!visited[v] && adjacencyMatrix[u][v] != 0 &&
                dist[u] != INT_MAX && dist[u] + adjacencyMatrix[u][v] < dist[v])
            {
                dist[v] = dist[u] + adjacencyMatrix[u][v];
            }
        }
    }

    printSolution(dist, given_start);
}
```

1. **Function Signature:** The **dijkstra** function takes the name of a starting vertex (**given_start**) as input and finds the shortest paths to all other vertices in the graph.

2. **Error Handling:** The function checks the validity of the given starting vertex by using **getVertexIndex(given_start)**. If the index is -1, indicating an invalid vertex, the function prints an error message and returns.

3. **Initialization:** The function initializes arrays **dist** and **visited** to keep track of the shortest distance from the source vertex and the visited vertices, respectively.

4. **Set Initial Distance:** The initial distance from the starting vertex to itself is set as 0, and all other distances are set to **INT_MAX**.

5. **Main Loop:** The function iterates **vertices - 1** times, finding the vertex with the minimum distance (not visited) in each iteration.

6. **Update Distances:** For each unvisited neighbor of the current vertex, it updates the distance if a shorter path is found.

7. **Print Solution:** After the algorithm completes, it calls **printSolution** to print the shortest distances.

8. **Graph Representation:** The graph is assumed to be represented using an adjacency matrix (**adjacencyMatrix**). The function uses **vertexNames** for printing vertex names.

9. **Utility Functions:** The code references two utility functions (**minDistance** and **printSolution**). These functions are assumed to be correctly implemented based on their names, and their functionality is not explicitly described in this summary.

Option 33: Apply Prim's Algorithm:

This option applies Prim's algorithm to find the minimum spanning tree in the graph.

```cpp
    // Now Prims Algorithm

    void Prims(string given_start)
    {
        int parent[vertices] = {0};
        int dist[vertices];
        bool visted[vertices];

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

        int startIndex = getVertexIndex(given_start);
        if (startIndex == -1)
        {
            cout << "Invalid starting vertex.Enter Valid Starting City" << endl;
            return;
        }

        dist[startIndex] = 0;

        for (int count = 0; count < vertices - 1; count++)
        {
            int u = minDistance(dist, visted);
            visted[u] = true;
            for (int j = 0; j < vertices; j++)
            {
                if (adjacencyMatrix[u][j] != 0 && visted[j] == false && adjacencyMatrix[u][j] < dist[j])
                {

                    parent[j] = u;
                    dist[j] = adjacencyMatrix[u][j];
                }
            }
        }

        printSolution1(parent, startIndex);
    }
};
```

1. **Function Signature:** The **Prims** function takes the name of a starting vertex (**given_start**) as input and finds the Minimum Spanning Tree.

2. **Initialization:** The function initializes arrays **dist**, **parent**, and **visited** to track the distances, parent vertices, and visited vertices, respectively.

3. **Error Handling:** It checks the validity of the given starting vertex by using **getVertexIndex(given_start)**. If the index is -1, indicating an invalid vertex, the function prints an error message and returns.

4. **Set Initial Distance:** The initial distance from the starting vertex is set to 0, and all other distances are set to **INT_MAX**.

5. **Main Loop:** The function iterates **vertices - 1** times, finding the vertex with the minimum distance (not visited) in each iteration.

6. **Update Distances and Parents:** For each unvisited neighbor of the current vertex, it updates the distance and parent if a shorter edge is found.

7. **Print Solution:** After the algorithm completes, it calls **printSolution1** to print the Minimum Spanning Tree.

8. **Graph Representation:** The graph is assumed to be represented using an adjacency matrix (**adjacencyMatrix**). The function uses **vertexNames** for printing vertex names.

9. **Utility Functions:** The code references two utility functions (**minDistance** and **printSolution1**). These functions are assumed to be correctly implemented based on their names, and their functionality is not explicitly described in this summary.

Option 0: Exit:
This option allows the user to exit the program.

```
        case 0:
            cout << "Exiting the program." << endl;
            break;
        default:
            cout << "Invalid choice!" << endl;
        }
    } while (choice != 0);
```

Conclusion

In conclusion, the Flight Dataset program offers a robust and versatile solution for the management and analysis of flight information. The program's menu-driven interface provides users with a user-friendly experience, allowing them to seamlessly navigate through the dataset and perform various operations efficiently.

The utilization of a doubly linked list for storing detailed flight information ensures a structured and organized representation of each flight's attributes, including crucial details such as departure and arrival times, carrier information, and more. This data structure facilitates easy insertion, deletion, and retrieval of flight records, enhancing the overall efficiency of the program.

Moreover, the incorporation of a graph for network analysis adds a layer of sophistication to the program, enabling users to gain valuable insights into the interconnections between different flights. The utilization of algorithms like Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm, and Prim's algorithm empowers users to explore relationships, find optimal paths, and uncover patterns within the flight network.

The program's ability to handle diverse queries, such as filtering flights based on specific criteria like carrier, origin, destination, and more, showcases its flexibility and adaptability to user requirements. Additionally, functionalities like finding flights with the highest departure or arrival delay, calculating average distances, and identifying the fastest flight contribute to the program's analytical capabilities.

In summary, the Flight Dataset program stands out as a comprehensive tool for both managing and gaining insights from flight data. Its well-designed interface, coupled with the strategic use of data structures and graph algorithms, positions it as an asset for aviation professionals, researchers, and enthusiasts seeking a deeper understanding of the intricate web of flight connections and attributes.