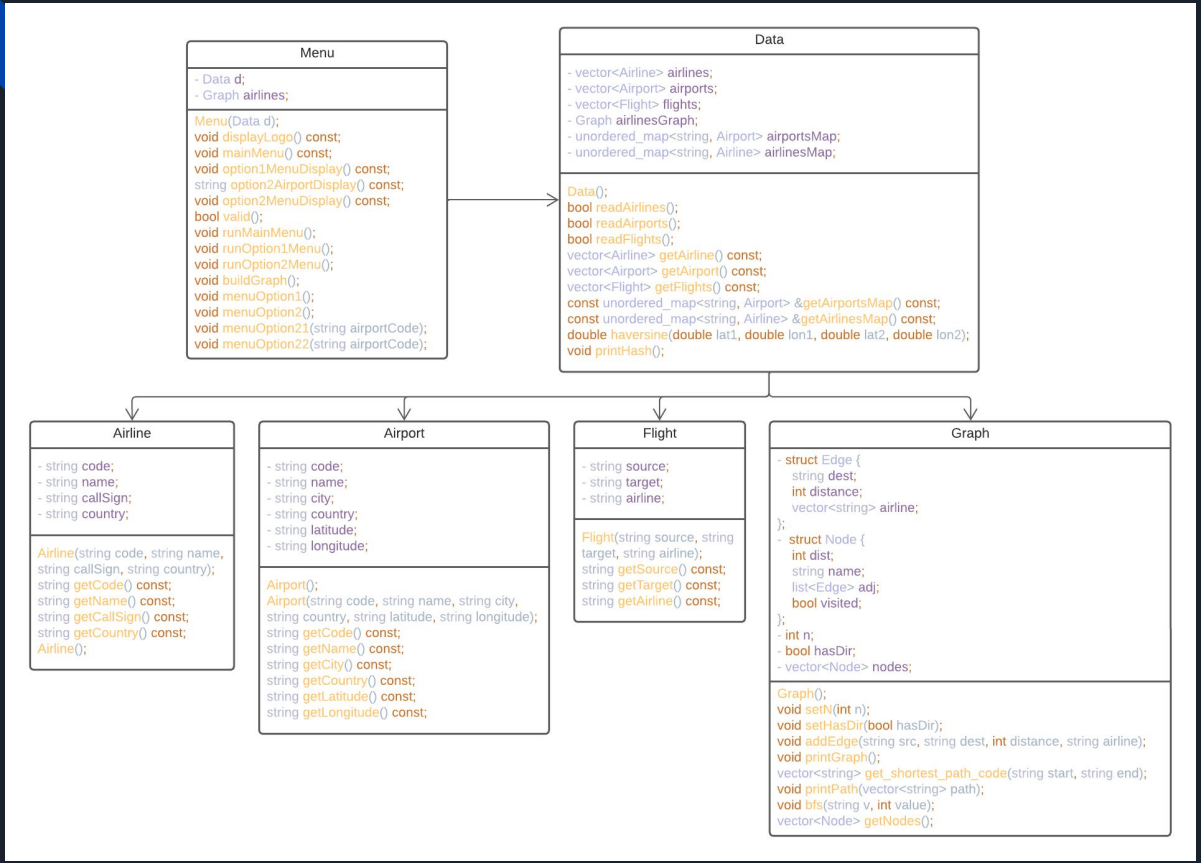
# Algoritmos e Estrutura de Dados

## TP2 - Transportes Aéreos


Grupo 106:

. Gonçalo Jorge Soares Ferreira - up202004761
. Martim Raúl da Rocha Henriques - up202004421
. Simão Queirós Rodrigues - up202005700

# Diagrama de classes




A classe *Data* envolve os 3 ficheiros fornecidos:

- airlines.csv
- airports.csv
- flight.csv

A classe Menu contém um atributo do tipo *Data*, onde estão guardadas as informações dos ficheiros, no qual realiza funções que procuram esses mesmos dados.

# Leitura do dataset

A leitura dos dados foi realizada de forma semelhante para os 3 ficheiros, podendo ser repartida em 3 partes essenciais:

- Abrir o ficheiro
- Obter uma string por linha do ficheiro e delimitar por vírgulas para cada argumento
- Criar um novo objeto do tipo desejado e adicionar ao vetor respetivo desse objeto

```
bool Data::readFlights() {
    fstream file_flights( s: "../Database/flights.csv");
    string line, word;
    if (!file_flights.is_open()) {
        cout << "Failed to open file\n";
        return false;
    }

    getline( &: file_flights, &: line);
    string target; string source; string airline;
    while(getline( &: file_flights, &: line)) {
        stringstream iss(line);
        getline( &: iss, &: target, delim: ',');
        getline( &: iss, &: source, delim: ',');
        getline( &: iss, &: airline, delim: ',');

        Flight flight( source: target, target: source, airline: airline);
        flights.push_back(flight);
    }

    file_flights.close();
    return true;
}
```

# Representação do dataset

Posteriormente é guardado no objeto *airlines*, atributo da classe *Menu*, segundo a função *buildGraph()*.

Este método começa por definir o tamanho do grafo e que este é direcionado,

Depois, para cada combinação entre a origem no vetor *flights* e o código do aeroporto adiciona um novo *Edge* no grafo onde se guarda a distância entre os 2 aeroportos.

```
void Menu::buildGraph() {
    airlines.setN( n: d.getFlights().size());
    airlines.setHasDir( hasDir: true);
    for (auto i : d.getFlights()) {
        double a1 = 0;
        double a2 = 0;
        double b1 = 0;
        double b2 = 0;
        for (auto j : d.getAirport()) {
            if (j.getCode() == i.getSource()) {
                a1 = stod( str: j.getLatitude());
                a2 = stod( str: j.getLongitude());
            }
            if (j.getCode() == i.getTarget()) {
                b1 = stod( str: j.getLatitude());
                b2 = stod( str: j.getLongitude());
            }
        }
        airlines.addEdge( src: i.getSource(), dest: i.getTarget()
    }
}
```

```
airlines.addEdge( src: i.getSource(), dest: i.getTarget(), distance: d.haversine(a1, a2, b1, b2), airline: i.getAirline());
```

# Representação do dataset

```
class Graph {
    struct Edge {
        string dest;   // Destination node
        int distance; // An integer weight
        vector<string> airline;
    };

    struct Node {
        int dist;
        string name;
        list<Edge> adj; // The list of outgoing edges (to adjacent nodes)
        bool visited;   // As the node been visited on a search?
    };

    int n;              // Graph size (vertices are numbered from 1 to n)
    bool hasDir;        // false: undirected; true: directed
    vector<Node> nodes; // The list of nodes being represented
```

# Funcionalidades implementadas

Origem/Destino:

- Todas as funcionalidades relacionadas com Origem/Destino estão implementadas, podendo realizar qualquer combinação entre aeroporto, cidade e coordenadas.

Rede de voos:

- Para cada uma das funcionalidades acima, o utilizador pode filtrar os seus resultados por uma ou mais companhias aéreas.

# Funcionalidades implementadas

A função *get_shortest_path_code* é uma adaptação do algoritmo de pesquisa BFS (Breadth First Search) usando unordered maps.

Como o nome indica, este método é utilizado para obter o caminho mais curto entre 2 pontos de referência em todas as funcionalidades da 1ª parte Origem/Destino.

O unordered_map *distance* é usado para guardar o nome dos aeroportos e a respetiva distância relativa ao aeroporto *start*.

O unordered_map *previous* é usado de forma a guardar o caminho até se encontrar o aeroporto *end*.

```
vector<string> Graph::get_shortest_path_code(string start, string end) {
    queue<Node> q;
    unordered_map<string, string> previous;
    unordered_map<string, int> distance;

    for (auto & node :Node& : nodes) {
        if (node.name == start) {
            q.push( x: node);
        }
    }

    distance[start] = 0;

    while (!q.empty()) {
        Node node = q.front();
        q.pop();
        if (node.name == end) {
            break;
        }
        for (auto & e :Edge& : node.adj) {
            if (distance.count( x: e.dest) == 0) {
                distance[e.dest] = distance[node.name] + 1;
                previous[e.dest] = node.name;
                for (auto & n :Node& : nodes) {
                    if (n.name == e.dest) {
                        q.push( x: n);
                    }
                }
            }
        }
    }

    vector<string> path;
    string current = end;
    while (current != start) {
        if (current.empty()) {
            return path;
        }
        path.insert( position: path.begin(), x: current);
        current = previous[current];
    }
    path.insert( position: path.begin(), x: start);
    return path;
}
```

# Funcionalidades implementadas

Informações sobre um aeroporto:

- Conseguimos implementar todas as funcionalidades relativas ao 2º objetivo. Nomeadamente, quantos voos existem a partir de um dado aeroporto, de quantas companhias aéreas diferentes, para quantos destinos diferentes, para quantos países diferentes e quantos aeroportos, cidades ou países são atingíveis usando um máximo de Y voos.

- Para cada uma destas funcionalidades existe uma opção de visualizar as informações completas de cada voo

# Destaque de funcionalidade

Decidimos destacar a funcionalidade presente no Menu.cpp como *option25().*

Realiza-se um bfs inicialmente para atribuir distâncias a todos os nós a partir do nó pretendido.

Onde calcula e imprime onde é possível chegar com um máximo de Y voos.

```
while (!q.empty()) { // while there are still unvisited nodes
    string u = q.front();
    q.pop();
    //cout << u << " "; // show node order
    for (auto node : nodes) {
        if (node.name == u) {
            if (node.dist > value)
                break;
            for (auto e: node.adj) {
                string w = e.dest;
                for (auto & j : nodes) {
                    if (j.name == w) {
                        if (!j.visited) {
                            q.push(w);
                            j.visited = true;
                            j.dist = node.dist + 1;
                        }
                    }
                }
            }
        }
```

# Dificuldades encontradas

Diminuir a complexidade dos nossos algoritmos foi a nossa maior dificuldade.

Participação dos membros: os três membros trabalharam e colaboraram igualmente no projeto.