Airplane Reservation System

Classes:

- Vector
- Date
- Time
- Node
- Adjlist
- ARS

Vector :

This class was made as an alternative for the build in class of Vector. This class basically has all the functionalities in addition with the property that whenever we want to add an item and the array is already filled, additional memory is automatically allocated to it.

Data Members:

- `T* ptr` — This is the main pointer which points towards the array of type T
- `int size` — This variable contains the number of cells whom the memory has been allocated or in easy language size of array
- `int index` — This variable contains the index after which no more data is inserted

Member Functions:

- `Vector()`
  This is the default constructor of Vector. In this the main pointer is set to null and index to -1. The size of array is set 0 as no array is being formed.
- `Vector(int n)`
  This is the parametrized constructor of Vector. In this the main pointer is allocated the memory of n cells and index is set to -1 because currently no data has been inserted. The size of array is set n.
- `~Vector()`
  This is the destructor of class in which we deallocate all the memory.
- `bool isEmpty()`
  This function checks whether the array is empty or not by checking the value of index.
- `void clear()`

This function clears the array by setting the value of index to -1 as a result of which the data starts inserting in array from the beginning of array.

- `void push(T data)`
  This function inserts the data at cell which is next to current cell and it also manages the situation if the array is filled. In that case it makes another array of double in size than the previous and adjusts all the situations properly.

- `T getAt(int i)`
  This function returns the value at the index sent as a parameter to the function.

- `int getSize()`
  This function returns the size of the array

- `int getNoofElements()`
  This function returns how many elements have been inserted in the vector.

- `T& operator[](int ind)`
  We have overloaded operator of "[]" so that we can use this class same as we do in arrays. This function returns the value at the "ind" (parametrized) index.

Time:

This class is basically formed to perform all the necessary operation on time. We needed time several times in our project during working on the times of flight that's why we made this class to deal with all those operations.

Data Memebers:

- `int hour` — This variable was used to store the no of hours in current time
- `int minutes` — This was used to store no of minutes in current time
- `string time_` — This was used to keep the data in proper format as a string so that we can display the

time of flights to the users in a understandable way

Member Functions:

- `Time()`
  This is the default constructor of this class in which we have set the values of hours and minutes to null and have set the string to “”.

- `void set_time(string t)`
  This is the parametrized constructor in which we have set the right value of the time and then we have set the value of hours and minutes by applying all the checks on string sand converting their values to integers by the help of ascii.

- `int check_time(const Time& obj)`
  This function checks what is relation between the parametrized time and time of its own class, whether it is small or not. If the function returns 1, it means that both of the times are exactly the same. If the function returns 2 it means that the time of our class is less than the parametrized class and the value of 3 signals that time of this class is greater than the parametrized class.

- `void add_time(const Time obj)`
  This function adds the time sent through the parameter to the current time.

- `void sub_time(const Time obj)`
  This function subtracts the time sent through the parameter from the original class of Time.

- `void update_time_str()`
  During performing many operations on this class the value of time changes so in order to always have the right value of time in the string this function is used.

- `void add_24Hours()`
  This function adds the 24 hours in current time which means that one day has elapsed

- `void sub_from_24Hour()`
  This function is used to subtract time from 24 hours so that we can get respective time of the next day

- `int get_hours()`

This function is used to get the number of hours

- int get_minutes()
  This funxtion is used to get the current number of minutes

- string time_to_str()
  This function returns the current value of time as a string so that we can display the proper format of the time

Date:

This class is basically formed to perform all the necessary operation on date. We needed date several times in our project during working on the dates of flight that's why we made this class to deal with all those operations.

Data Memebers:

- int day — This variable was used to store the no of days in current date e.g. for 8/12/2019 it will store 8 in days
- int month — This was used to store no of months in current date e.g. for 8/12/2019 it will store 12 in months
- int year — This was used to store current year in date e.g. for 8/12/2019 it will store 2019 in year
- string date_ — This was used to keep the data in proper format as a string so that we can display the dates of flights to the users in a understandable way

Member Functions:

- Date()
  is the default constructor of this class in which we have set the values of days, months and year to null and have set the string to "".

- void set_Date(string dt)
  This is the parametrized constructor in which we have set the right value of the date and then we have set the value of days, months and year by applying

all the checks on string sand converting their values to integers by the help of ascii.

- `int compare_date(const Date& obj)`
  This function checks what is relation between the parametrized date and date of its own class, whether it is small or not. If the function returns 1, it means that both of the dates are exactly the same. If the function returns 2 it means that the date of our class is less than the parametrized class and the value of 3 signals that date of this class is greater than the parametrized class.

- `int get_Day()`
  This function returns the current value of the day

- `int get_Month()`
  This function returns the current value of month

- `int get_Year()`
  This function returns the current value of year

- `int compare_date(const Date& obj, int val)`
  This function adds the value in the day of current date and then checks the relation between both the classes that whether they are equal or less than or greater than just like the function of compare classes

- `string to_str()`
  This function returns the current value of date as a string so that we can display the proper format of the date

Node:

This class is used as vertex for the graph. It will save all the information about the flight. Its origin, destination, flying time, landing time, date of flight, airplane etc.

Data Memebrs:

- `string Origin;` — ciy of origin
- `Node* Destination;` — a pointer pointing to the city of destination

- `int TicketPrice;` — ticket Price for flight
- `string Airline;` — name of Airline
- `int Hotel_Charg;` — hotel charges of that country
- `Date DateOfTravel;` — date of travel of flight
- `Time FlyingTime;` — flying time of flight
- `Time LandingTime;` — landing time of flight
- `Time Travel_Time;` — total time of the flight
- `Time Transit_Time;` — transit time between this flight and the next flight
- `bool p_check;` — if path is valid p_check is true else false

Member Functions:

- `Node()`
  Default Constructor. It sets all varaibles to default values
- `void show_data()`
  This function displays the data of node

AdjList:

This class is used to create a dynamic list of nodes.It is used to create the Adjancy list in the graph.

Data Memebers:

- `Node* head;` — head of the list

Member Functions:

- AdjList()
  Default Constructor sets head to null

- void insert(Node* newDataItem)
  This fucntion add the newDataItem in the list

Working:
If head is null the newDataItem is added to head. Otherwise we go the end of the list using Destination and attached the newDataItem to the end of the list.

- void display()
  This function is used to display the contents of list.

Working:
A temporary variable points to starting node of the list. Until varaible does not becomes null the contents which varaible is pointing are dislpayed and varaible is updates to its destination. In this way whole list is displayed. It takes O(n) time where n is nodes in the list.

- bool search(string value)
  This function searches a string value in the list

Working:
A boolean varaible found is set to zero. Using a temporary varaible, which initally points to the first node in the list, we traverse through the list. If any node's Origin is the same as searching value

ARS:

This is the main class for our airplane reservation system.It contains an adjancy list in the form of array variable. Cities, dates, airlines, hotel charges arrays are made to make searching easy. Number of dates, airlines, cities are also made to decrease the time complexity. So that we can easily search desired value.

Data Memebers:

- AdjList* array; — array of adjoincy list
- int NoOfCities; — variable to store no of cities
- int number — no of paths

- `string* cities;` array of cities anmes
- `string* dates;` array of dates
- `string* airlines;` array of airlines
- `int* hotel_charges;` array of hotel charges
- `int num_dates;` number of dates
- `int num_airlines;` number of airlines

Member Functions:

- `ARS(string flights, string hotels)`
  This is the main constuctor of ARS class

Working:

First using the string in the parameters of the constructore fstream type objects are created to read files. Some temporary vectors are made to save the city names, airline names and dates. Then from the flights file only city names, airline names and dates are read and saved in some vectors. Then these names are saved in arrays. Number of cities/ vertices of graph are determined by the number of cities in the vector. Also using Number of cities array of graph is intialized. Then using hotels charges file hotel charges are read. Corresponding city is searched in the array of cities and using the index of that city, hotel charges are saved in the hotel charges array. Starting vertices are added in graph using city names array.

Then from the start of the flights file, origin, destination, date of flight, flying time, landing time, ticket price and airline name are read. A nod eis crreated using the above read data. Travel is calculated on the bases of flying and landing time. If landing time is greater than the flying time then total travel time is landing time – flying time of flight. If Landing time is less than the flying time it means the flight is going to land on the next day so the travel time is landing time + 24 hours – Flying time. Then hotel charges for that city are added in the node.

Then the index of Adjancy list to which it has to be attached is searched. A node is checked for duplication. If it is already in the list it is rejected else it is added to Adjancy list.

- `AdjList* get_array()`
  This function returns the array / adjancy list array. It takes O(1) time

- `int Num_Cities()`
  This fucntion returns the number of cities/ vertices in graph. It takes O(1) time

- `string* CityNames()`
  This function returns pointer of array of city names. It takes O(1) time

- `string* Dates_A()`
  This function returns pointer of array of city dates. It takes O(1) time

- `string* Airlines_Names()`
  This function returns pointer of array of airlinesnames. It takes O(1) time

- `int Numb_Dates()`
  This function returns total number of avalible dates. It takes O(1) time.

- `int Numb_Airline()`
  This function returns total number of airlines. It takes O(1) time.

- `Vector<Vector<Node>> Get_Paths(string source_, string destination_)`
  This Function returns all the possible paths from source node to the destination in the form of Vectors of path

Working:

This function first uses the source and destination strings to search for nodes with similar city names in the Adjancy list. Once the index number is found, that index number is used to create two nodes and their data is the same as the node that was found with similar city. The searching will take O(n) time, where n is total number of cities / vertices in the graph.

Then a boolean array of vistited is created. This is used to check later whether a node was visited or not. A node type array named path is used to save nodes that are in the path. Both these arrays have size equal to the total number of cities in graph because as no city can be repeated in path to avoid loop so only maximum path can be a path whose length is equal to number of cities. Initially all the elements of vistied array are set to 0 or in other words they all are not vistited.

There is a paths varaible which will be saving all the possible paths. It is basically a Vector of Vector or Vector of path, where path will be a Vector of Nodes. We have choose Vectors as our data structure for paths because if Vectors are dynamic. Their elements can be accessed using [ ] operator. It is easy to traverse Vector using the for loop. Also any element can be acessed using index as compared other data structures like list where we would have to check for next pointer untill we donot find our desired value or node.

Then All Paths function is called which uses depth first search to go through all the nodes in the graph starting from source node. When All Paths function is executed paths contains all the possible paths from souce to destination.

- `void Total_Time_And_Charges(Vector<Vector<Node>>& paths)`
  This function calculates the total time of flight and the total charges for the flight and saves in the first node of each path

Working:

A for loop goes through all the paths. And for each indiviual path, if it is not the first node then the ticket price is added to ticket price of first node and

the travel time is added to the travel time of first node. Here to save space we are using the first node as a representative of whole path for total travel time and total fare of the whole journey. If the node is not the first and last node then transit time is calculated for each node.

If landing time of flight is greater than the flying time i.e the journey starts and end on the same day then if the next flight is on the same day the transit time is calculated as flying time of next flight – landing time of flight 1. Other wise for different day transit time is calculated as (remaining hours of day i.e 24 hours in day – landing time of flight 1) + for each next day untill the date of next flight is not reached a 24 hour is added to transit time + flying time of the next flight.

If the landing time is less than the flying time of the flight then it means that the flight will land on the next day. In this case for each day untill the date of next flight is not reached a 24 hours is added to transit time. When the date is reached, the flying time of next flight is added and landing time of flight 1 is subtracted from transit time.

Transit time is saved in the node and also added to the travel time of first node. If the transit time is greater than 12 hours hotel charges are added in the ticket price of the first node in the path. This whole takes O(n-2) time because first and last nodes in each path are skipped for transit hours.

- `void Print_Paths(Vector<Vector<Node>>& paths)`

  This Function prints all the paths if valid

Working:

A for loop goes through each path. If the path is valid (it checks the p_check of first node of each path) then it prints all the nodes in the path. It takes O(n) to check path. If found then takes 0(n) to print them.

- `void All_Paths(Node* u, Node* d, bool visited[], Node path[], int& path_index, Vector<Vector<Node>>& paths)`

  This function uses the principle of depth first serach to go to all nodes from source and add those paths which end at destination and are valid, to paths

Working:

First it finds the index of the source nod ein the Adjancy list and mark the position of the source city as true in the visited. This take O(n) time. Then it add that node to the path. As path_index is saving the number of nodes in the path so it is incremented.

If Destination is found:

If destination is found then it is checked whether the path is correct is not. Correctness of path depands on many factor. Flight 2 can not start before the end of the Journey of flight 1. Also the Flight 2 can not be on the day before the date of flight of Flight 1. So for this we traverse all the nodes in the path. We skip the any node if it does not contain the date of travel, time etc (that can be the case for first node) and also the last node (because there is no node after that to compare

it with). Then date of travel of next flight is checked. If it Flight 1's date of travel is after the Date of travel for Flight 2 then it is marked as not a valid path.

If Flight 1' date of travel is less than the date oftravel for flight 2 then other conditions are checked. If they are on the same day then: 1) if the flying time of flight 1 is greater than landing time then path is not valid because then it might be landing on next day and next flight will be after that, not on same day.
2) if it is not the previous case, then if the flying time of flight 2 is less than the landing time of flight 1 then it will also be an invalid path.

If flight 2 is on next day of flight 1 then if flight 1 was landing on next day then flying time of flight 2 should be greater than the landing time of flight 1.

If after all these conditions the path is valid then each node of path is added to a vector and the vector is added to paths.

If Destination was not found:

Then go to each next node of souce (skipping the souce itself). Check if that node was already vistied or not. If it was not visited then call function All Paths with this new child node as souce.
Remove the last node from path and mark it as unvistied.

- ```
  void Show_Routes(string origin, string destination, string date,
              bool transit_loc_b, string transit_loc,
              bool transit_hours_b, string min, string max,
              bool airline_b, string airline,
              bool direct_flight, int cost_or_time, bool again)
  ```

This function gets all the possible paths from souce to destination with user preferences and displays them

Working:

First if user does not specify any prefernce for time or cost optimization then time is selected as a default option. Then all possible paths are get using the Get Paths function and saved in a variable named paths. Total Travel time and total charges for flight are calculated using the Total Time And Charges function. Then using Sort Paths function and the prefrence of user, if specified otherwise time, paths are sorted. Check Date function is used to check if paths have correct date. Afterwards if a transit location is specified then it is checked that the path contains the transit location. If a transit hours are also specified then they are also checked. If user has specified an airline preference then it is also checked. In case the user has not selected any transit location but has selected then direct flights option then only direct flights are selected in paths.

Starting date is displayed along with paths. If no paths were found then flights of a day before and after are displayed.
  ```

- `void Check_Transit_Location(Vector<Vector<Node>>& paths, string transit_loc)`
  This function checks if the paths contain the transit location or not

Working:

A for loop is used to traverse through each path. Another for loop is used to used traverse through the nodes in path. Leaving first and last nodes in the path, it it checked if at any point in path the node's origin is the same as transit location. If they match then a boolean varaible is set to true. After the completion of the inner loop if the boolean variable if set to 0 then it means no transit location was found and path is set as invalid path.

- `void Check_Direct_Flights(Vector<Vector<Node>>& paths)`
  This function checks for direct flights in the paths

Working: Using a for loop the length of each path is checked if it is greater than two the path is marked as invalid. It takes O(n) time.

- `void Check_Airline(Vector<Vector<Node>>& paths, string airline)`
  This function searches if a path a the specified airline or not

Working:

A for loop is used to traverse through paths. Another for loop is used to check if the nodes contain that airline or not. If the path does not have that airline it is marked as invalid else if it valid.

- `int Num_Paths(Vector<Vector<Node>>& paths)`
  This Function returns the number of paths that are valid

Working:

A count varaible is used to maintain a count of paths. Using a for loop it is checked if a path's first node's p_check is true or not. If it is true then path is valid and count is incremented. In the end count is returned. It takes O(n) time.

- `void Check_Transit_Hours(Vector<Vector<Node>>& paths, string transit_loc, string min, string max)`
  This function checks if the paths contain the transit location with valid transit stay or not

Working:

A for loop is used to traverse through each path. Another for loop is used to used traverse through the nodes in path. Leaving first and last nodes in the path, it it checked if at any point in path the node's origin is the same as transit location. If they matched then the transit time is checked if transit time greater than or equal to minimum transit time and transit time is less than or equal to maximum transit time. If transit time is correct then a boolean varaible is set to 1. At the end of the inner loop if that boolean is found to 1 then it is a valid path else it is marked as an invalid path.

- `void Check_Date(Vector<Vector<Node>>& paths, string date)`

This function checks the paths if the start of path has correct date

Working:
A for loop goes through each path. If the 2nd node of each path has date as specified then it is marked as a valid path else an invalid path. It takes O(n) time.

- `void Sort_Paths(Vector<Vector<Node>>& paths, int cost_or_travel)`
  This function sorts the paths according to the cost or travel time

Working:
A for loop goes through n – 1 paths in the paths vector. Then leaving the number of paths as estimated by the outer loop it traverse through each path. If cost is specified then it checks the ticket price of first node in the path else it checks the travel time of first node. If it is greater than the ticket price or travel time of next path then it swaps them else it does nothing.

Some Extra Functions:

`int find_value(string arr[], string value, int size)`
This function searches a string in the array and returns its index number is found.

Working:
A variable index is initially set to -1. We traverse through the array using a for loop and check if the value in array is equal to to searched value. If value is found the loop is terminated and position is saved in index. It takes O(n).

`bool search_string(Vector<string>& t, string str)`
This function searches the string in the vector and returns true or false
Working: Using a for loop we traverse through the whole vector and check if the value matches the specified search value. If they then true is returned else false if returned