Assignment 2

Submitted By : Mushtasin Ahmed
Id : 2322175742
Section : 18
Course : CSE225

Submitted to: Mr. Rifat Ahmed Hassan [RIH]

Q. You want to store the stations of a public transportation line. New stations can be added to both ends of the line, but not between existing stations.You should be able to traverse the line in both directions.

Since we are adding stations from both sides , So Doubly linked list would be perfect in this case . As it will allow access to nodes from both sides (Considering each station as a node) Making the traversal from both sides easier .

Justification:
Since our requirement is to add new stations only to both ends of the line (not in between existing stations). And also have the ability to traverse the line in both directions.

- Efficient Insertion at both ends : O(1)

Since we hold pointers to the head and tail so the insertion can be achieved within constant time of O(1)

- No Fixed Size:

Since we don't know the number of stations , we cant have a fixed size. That's why Doubly Linked List being a dynamic data structure,we don't need to worry about the capacity or resizing, unlike arrays.

- Achieving Bidirectional Traversal:

As each node is holding pointers to the previous and next node, this makes it easy to traverse the line in both directions. In our implementation each node contains two pointers, one pointing to the next station and one pointing to the previous station.

Q. You are writing software for a call center. When a client calls, his call should be stored until there is a free operator to pick it up. Calls should be processed in the same order they are received.

-Since we are required to store the calls until there is a free operator. And also calls should be processed in the order they are received (FIFO: First-In, First-Out).
So the best-suited Data Structure is: Queue

Justification :

- FIFO Order:

A queue follows the first in first out order which accurately matches with our requirement to handle calls in the sequence they are received.

- Efficient Operations:

Time complexity for Enqueuing (adding a call to the queue) is done in constant time that is O(1) time.
Also Time complexity for dequeuing (removing a call when an operator is available) take O(1) time.

Hence The queue data structure is the optimal choice due to its simplicity, natural alignment with the FIFO processing order, and efficient O(1) operations for both enqueuing and dequeuing.

Q2 . Create Airline Ticket Reservation System

We shall use **Linked Lists** to manage flights and passengers. We'll organize the solution as follows:

1. **A Linked List for Flights**: Each node represents a flight and contains:
    - The flight number.
    - A linked list of passengers.
2. **A Linked List for Passengers** : Each node represents
    - The passenger's name.
    - Next Pointer

Data Structures Chosen:

Singly Linked List for Flights: This allows for efficient traversal and dynamic addition of new flights.

Singly Linked List for Passengers: Used within each flight node to maintain the list of passengers. The list We will efficiently manage flights and passengers through the use of Linked Lists, structured in the following manner:

Reasoning behind our chosen Data Structure:

Linked Lists

Since they provide dynamic memory allocation, so we can easily add or remove flights and passengers without worrying about resizing. The trade-off is linear search time (O(n)) to find flights or passengers, which is acceptable for relatively small lists.

Also we don't know the number of passengers beforehand so creating an array isn't that logical .

Also Alphabetical Sorting of passengers is done by inserting passengers into their correct position within the linked list, maintaining their respective order.

The choice of Linked Lists allows for dynamic memory allocation, which is advantageous as it eliminates concerns about resizing arrays. Although using a linked list means that searches for flights or passengers will require linear

time complexity (O(n)), this is acceptable given that we are managing relatively small collections of data.

CODE

File: “Airline-reservation-system.h”

```
#ifndef AIRLINE_RESERVATION_SYSTEM_H
#define AIRLINE_RESERVATION_SYSTEM_H
#include <iostream>
#include <string>

using namespace std;

// Node for the Passenger Linked List
 class PassengerNode{
   public:
   string name;
   PassengerNode* next;

   PassengerNode(const string& passengerName){
       this->name=passengerName;
       this->next=nullptr;
};

// Node for the Flight Linked List
struct FlightNode {
    string flightNumber;
    PassengerNode* passengerList;
    FlightNode* next;

    FlightNode(const string& flightNum){
        this->flightNumber=flightNum;
        this->passengerList=nullptr;
        this->next=nullptr;

    };

};
```

```
// Airline Reservation System Class
class AirlineReservationSystem {
private:
    FlightNode* flightList; // Head of the Flight Linked List
    FlightNode* findFlight(const string& flightNumber);
    void addPassenger(PassengerNode*& head, const string& passengerName);
    bool deletePassenger(PassengerNode*& head, const string& passengerName);
    void displayPassengers(PassengerNode* head) const;
public:
     AirlineReservationSystem();
    ~AirlineReservationSystem();
 void reserveTicket(const string& flightNumber, const string& passengerName);
 void cancelReservation(const string& flightNumber, const string&
passengerName);
 void checkReservation(const string& flightNumber,const string& passengerName)
;
 void displayPassengers(const string& flightNumber);
};
#endif // AIRLINE_RESERVATION_SYSTEM_H
```

File - “main.cpp”

```
#include <iostream>
#include <string>
#include "Airline-reservation-system.h"
using namespace std;

// Airline Reservation System class
class AirlineReservationSystem {
private:
    FlightNode* flightList; // Head of the flight linked list
    // Function to find a flight
    FlightNode* findFlight(const string& flightNumber) {
        FlightNode* temp = flightList;
```

```
        while (temp) {
            if (temp->flightNumber == flightNumber) {
                return temp;
            }
            temp = temp->next;
        }
        return nullptr;
    }

    // function to find and insert a passenger in alphabetical order
    void addPassenger(PassengerNode*& head, const string&
passengerName) {
        PassengerNode* newPassenger = new
PassengerNode(passengerName);
        if (!head || head->name > passengerName) {
            newPassenger->next = head;
            head = newPassenger;
        } else {
            PassengerNode* current = head;
            while (current->next && current->next->name < passengerName)
{
                current = current->next;
            }
            newPassenger->next = current->next;
            current->next = newPassenger;
        }
    }
```

```
    // function to delete a passenger
    bool deletePassenger(PassengerNode*& head, const string&
passengerName) {
        if (!head) return false;

        if (head->name == passengerName) {
            PassengerNode* temp = head;
            head = head->next;
            delete temp;
            return true;
        }

        PassengerNode* current = head;
        while (current->next && current->next->name != passengerName) {
            current = current->next;
        }

        if (current->next) {
            PassengerNode* temp = current->next;
            current->next = current->next->next;
            delete temp;
            return true;
        }
        return false;
    }
```

```
    // Function to display passengers
    void displayPassengers(PassengerNode* head) const {
        while (head) {
            cout << "- " << head->name << endl;
            head = head->next;
        }
    }

public:
    AirlineReservationSystem() : flightList(nullptr) {}

    // Reserve a ticket for a passenger on a flight
    void reserveTicket(const string& flightNumber, const string&
passengerName) {
        FlightNode* flight = findFlight(flightNumber);
        if (!flight) {
            flight = new FlightNode(flightNumber);
            flight->next = flightList;
            flightList = flight;
        }
        addPassenger(flight->passengerList, passengerName);
        cout << "Ticket reserved for " << passengerName << " on flight " <<
flightNumber << ".\n";
    }
```

```
    // Cancel a reservation for a passenger on a flight
    void cancelReservation(const string& flightNumber, const string&
passengerName) {
        FlightNode* flight = findFlight(flightNumber);
        if (flight) {
            if (deletePassenger(flight->passengerList, passengerName)) {
                cout << "Reservation for " << passengerName << " on flight "
<< flightNumber << " canceled.\n";
            } else {
                cout << "No reservation found for " << passengerName << " on
flight " << flightNumber << ".\n";
            }
        } else {
            cout << "Flight " << flightNumber << " not found.\n";
        }
    }

    // Check if a passenger has a reservation on a flight
    void checkReservation(const string& flightNumber, const string&
passengerName) {
        FlightNode* flight = findFlight(flightNumber);
        if (flight) {
            PassengerNode* passenger = flight->passengerList;
            while (passenger) {
                if (passenger->name == passengerName) {
                    cout << passengerName << " has a reservation on flight "
<< flightNumber << ".\n";
                    return;
                }
                passenger = passenger->next;
```

```
            }
            cout << passengerName << " does not have a reservation on
flight " << flightNumber << ".\n";
        } else {
            cout << "Flight " << flightNumber << " not found.\n";
        }
    }

    // Display all passengers on a flight
    void displayPassengers(const string& flightNumber) {
        FlightNode* flight = findFlight(flightNumber);
        if (flight) {
            if (flight->passengerList) {
                cout << "Passengers on flight " << flightNumber << ":\n";
                displayPassengers(flight->passengerList);
            } else {
                cout << "No passengers on flight " << flightNumber << ".\n";
            }
        } else {
            cout << "Flight " << flightNumber << " not found.\n";
        }
    }
};
```

```
// Main Function
int main() {
    AirlineReservationSystem system;
    int choice;
    string flightNumber, passengerName;

    do {
        cout << "\nAirline Ticket Reservation System\n";
        cout << "1. Reserve a ticket\n";
        cout << "2. Cancel a reservation\n";
        cout << "3. Check if a ticket is reserved for a person\n";
        cout << "4. Display passengers on a flight\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        if(choice==1){
            cout << "Enter flight number: ";
            cin >> flightNumber;
            cout << "Enter passenger name: ";
            cin.ignore();
            getline(cin, passengerName);
            system.reserveTicket(flightNumber, passengerName);
            break;
        }else if(choice ==2){
             cout << "Enter flight number: ";
            cin >> flightNumber;
            cout << "Enter passenger name: ";
            cin.ignore();
            getline(cin, passengerName);
```

```
            system.cancelReservation(flightNumber, passengerName);
            break;
        }else if(choice==3){
             cout << "Enter flight number: ";
            cin >> flightNumber;
            cout << "Enter passenger name: ";
            cin.ignore();
            getline(cin, passengerName);
            system.checkReservation(flightNumber, passengerName);
            break;
        }else if(choice==4){
            cout << "Enter flight number: ";
            cin >> flightNumber;
            system.displayPassengers(flightNumber);
            break;
        }else if(choice==5){
            cout << "Exiting the program.\n";
            break;
        }else{
            cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 5);

    return 0;
}
```

Explanation of Code and its Workflow :

To handle flight reservations, the airline reservation system makes use of linked lists. In addition to displaying people on a certain aircraft, it facilitates functions such as booking a ticket, canceling a reservation, and determining if a person has a reservation. Flights are recorded in another linked list, and each flight is represented by a node holding a linked list of passengers. The system effectively manages dynamic activities, enabling the addition or cancellation of bookings without a set size restriction. For example, adding a passenger or flight takes linear time in relation to the size of the list.

An empty list of flights is initialized at the start of the airline reservation system. The user is shown a menu with choices to display passengers on a flight, verify if a person is reserved, cancel a reservation, and reserve a ticket. It looks for the flight when a reservation is made; if it cannot be found, a new flight is made and the passenger is added. While the show option lists every passenger on a given aircraft, the check option searches the flight and its passenger list. Until the user decides to quit, the driver code keeps on running.