Data Structures & Algorithms

CSE225

ASSIGNMENT 02

SUMMER 2023

Submitted By:
MD. MISBAH KHAN
ID: 2132089642

Submitted To:
RIFAT AHMED HASSAN

Section 04

Answer to question 01:

- **Scenario 1:** Array, because in a dynamically allocated array, we can add as many stations as we want. Now as we don't need to insert any station at the middle, so we will only work with start and end. The starting index of an array is always 0. And the end index is total station number – 1. That's how we can add station at both end and traverse both direction by increasing or decreasing index.

- **Scenario 2:** Queue, because it follows First-In-First-Out (FIFO) principle. So, who called first, his call will be answered first, if one or more call come during handling the first one, they will be queued one after another following FIFO principle.

Answer to question 02:

It will be best if we use BST for this this problem. Because we have total 4 operations to do in this problem. Of which, using BST, Time complexity will be:

Case 1: Inserting a flight: **O(log N).**
Case 2: Searching for a flight: **O(log N).**
Case 3: Searching for a passenger(throughout all flights): **O(N logN).**
Case 4: Displaying the passengers list: **O(N).**

So, the overall Time Complexity will be **O(N logN).**

Here is the code for this problem. In this code, we use **binarysearchtree.h, binarysearchtree.cpp, quetype.h, quetype.cpp** from the slide discussed in class. The driver file (main.cpp) is modified by me.

::::::::::::::quetype.h::::::::::::::

```
#ifndef QUETYPE_H_INCLUDED
#define QUETYPE_H_INCLUDED

class FullQueue {
};

class EmptyQueue {
};

template<class ItemType>
class QueType {
    struct NodeType {
        ItemType info;
        NodeType *next;
```

```
    };

    public:
        QueType();
        ~QueType();
        void MakeEmpty();
        void Enqueue(ItemType);
        void Dequeue(ItemType &);
        bool IsEmpty();
        bool IsFull();

    private:
        NodeType *front, *rear;
};

#endif // QUETYPE_H_INCLUDED
```

::::::::::::::quetype.cpp::::::::::::::

```
#include <iostream>
#include "quetype.h"
using namespace std;

template<class ItemType>
QueType<ItemType>::QueType() {
    front = NULL;
    rear = NULL;
}

template<class ItemType>
bool QueType<ItemType>::IsEmpty() {
    return (front == NULL);
}

template<class ItemType>
bool QueType<ItemType>::IsFull() {
    NodeType *location;
    try {
        location = new NodeType;
        delete location;
        return false;
    }
    catch (bad_alloc &exception) {
        return true;
    }
}

template<class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem) {
    if (IsFull())
```

```
        throw FullQueue();
    else {
        NodeType *newNode;
        newNode = new NodeType;
        newNode->info = newItem;
        newNode->next = NULL;
        if (rear == NULL)
            front = newNode;
        else
            rear->next = newNode;
        rear = newNode;
    }
}

template<class ItemType>
void QueType<ItemType>::Dequeue(ItemType &item) {
    if (IsEmpty())
        throw EmptyQueue();
    else {
        NodeType *tempPtr;
        tempPtr = front;
        item = front->info;
        front = front->next;
        if (front == NULL)
            rear = NULL;
        delete tempPtr;
    }
}

template<class ItemType>
void QueType<ItemType>::MakeEmpty() {
    NodeType *tempPtr;
    while (front != NULL) {
        tempPtr = front;
        front = front->next;
        delete tempPtr;
    }
    rear = NULL;
}

template<class ItemType>
QueType<ItemType>::~QueType() {
    MakeEmpty();
}
```

::::::::::::binarysearchtree.h::::::::::::

```
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED

#include "quetype.h"

template<class ItemType>
struct TreeNode {
    ItemType info;
    TreeNode *left;
    TreeNode *right;
};
enum OrderType {
    PRE_ORDER, IN_ORDER, POST_ORDER
};

template<class ItemType>
class TreeType {
    public:
        TreeType();
        ~TreeType();
        void MakeEmpty();
        bool IsEmpty();
        bool IsFull();
        int LengthIs();
        ItemType RetrieveItem(ItemType& item, bool& found);
        void InsertItem(ItemType item);
        void DeleteItem(ItemType item);
        void ResetTree(OrderType order);
        void GetNextItem(ItemType& item,OrderType order, bool& finished);
        void Print();

    private:
        TreeNode<ItemType>* root;
        QueType<ItemType> preQue;
        QueType<ItemType> inQue;
        QueType<ItemType> postQue;
};

#endif // BINARYSEARCHTREE_H_INCLUDED
```

::::::::::::::binarysearchtree.cpp::::::::::::::

```
#include <iostream>
#include "binarysearchtree.h"
#include "quetype.cpp"

using namespace std;

template <class ItemType>
TreeType<ItemType>::TreeType()
{
    root = NULL;
}

template <class ItemType>
void Destroy(TreeNode<ItemType> *&tree)
{
    if (tree != NULL)
    {
        Destroy(tree->left);
        Destroy(tree->right);
        delete tree;
        tree = NULL;
    }
}

template <class ItemType>
TreeType<ItemType>::~TreeType()
{
    Destroy(root);
}

template <class ItemType>
void TreeType<ItemType>::MakeEmpty()
{
    Destroy(root);
}

template <class ItemType>
bool TreeType<ItemType>::IsEmpty()
{
    return root == NULL;
}

template <class ItemType>
bool TreeType<ItemType>::IsFull()
{
    TreeNode<ItemType> *location;
    try
    {
        location = new TreeNode<ItemType>;
        delete location;
        return false;
    }
    catch (bad_alloc &exception)
```

```
    {
        return true;
    }
}

template <class ItemType>
int CountNodes(TreeNode<ItemType> *tree)
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) +
               CountNodes(tree->right) + 1;
}

template <class ItemType>
int TreeType<ItemType>::LengthIs()
{
    return CountNodes(root);
}

template <class ItemType>
void Retrieve(TreeNode<ItemType> *tree, ItemType &item, bool &found)
{
    if (tree == NULL)
        found = false;
    else if (item < tree->info)
        Retrieve(tree->left, item, found);
    else if (item > tree->info)
        Retrieve(tree->right, item, found);
    else
    {
        item = tree->info;
        found = true;
    }
}

template <class ItemType>
ItemType TreeType<ItemType>::RetrieveItem(ItemType &item, bool &found)
{
    Retrieve(root, item, found);
    if (!found)
    {
        return ItemType();
    }
    return item;
}

template <class ItemType>
void Insert(TreeNode<ItemType> *&tree, ItemType item)
{
    if (tree == NULL)
    {
        tree = new TreeNode<ItemType>;
        tree->right = NULL;
        tree->left = NULL;
```

```
        tree->info = item;
    }
    else if (item < tree->info)
        Insert(tree->left, item);
    else
        Insert(tree->right, item);
}

template <class ItemType>
void TreeType<ItemType>::InsertItem(ItemType item)
{
    Insert(root, item);
}

template <class ItemType>
void Delete(TreeNode<ItemType> *&tree, ItemType item)
{
    if (item < tree->info)
        Delete(tree->left, item);
    else if (item > tree->info)
        Delete(tree->right, item);
    else
        DeleteNode(tree);
}

template <class ItemType>
void DeleteNode(TreeNode<ItemType> *&tree)
{
    ItemType data;
    TreeNode<ItemType> *tempPtr;
    tempPtr = tree;
    if (tree->left == NULL)
    {
        tree = tree->right;
        delete tempPtr;
    }
    else if (tree->right == NULL)
    {
        tree = tree->left;
        delete tempPtr;
    }
    else
    {
        GetPredecessor(tree->left, data);
        tree->info = data;
        Delete(tree->left, data);
    }
}

template <class ItemType>
void GetPredecessor(TreeNode<ItemType> *tree, ItemType &data)
{
    while (tree->right != NULL)
        tree = tree->right;
    data = tree->info;
}
```

```
template <class ItemType>
void TreeType<ItemType>::DeleteItem(ItemType item)
{
    Delete(root, item);
}

template <class ItemType>
void PreOrder(TreeNode<ItemType> *tree, QueType<ItemType> &Que)
{
    if (tree != NULL)
    {
        Que.Enqueue(tree->info);
        PreOrder(tree->left, Que);
        PreOrder(tree->right, Que);
    }
}

template <class ItemType>
void InOrder(TreeNode<ItemType> *tree, QueType<ItemType> &Que)
{
    if (tree != NULL)
    {
        InOrder(tree->left, Que);
        Que.Enqueue(tree->info);
        InOrder(tree->right, Que);
    }
}

template <class ItemType>
void PostOrder(TreeNode<ItemType> *tree, QueType<ItemType> &Que)
{
    if (tree != NULL)
    {
        PostOrder(tree->left, Que);
        PostOrder(tree->right, Que);
        Que.Enqueue(tree->info);
    }
}

template <class ItemType>
void TreeType<ItemType>::ResetTree(OrderType order)
{
    switch (order)
    {
    case PRE_ORDER:
        PreOrder(root, preQue);
        break;
    case IN_ORDER:
        InOrder(root, inQue);
        break;
    case POST_ORDER:
        PostOrder(root, postQue);
        break;
    }
}
```

```
template <class ItemType>
void TreeType<ItemType>::GetNextItem(ItemType &item, OrderType order, bool &finished)
{
    finished = false;
    switch (order)
    {
    case PRE_ORDER:
        preQue.Dequeue(item);
        if (preQue.IsEmpty())
            finished = true;
        break;
    case IN_ORDER:
        inQue.Dequeue(item);
        if (inQue.IsEmpty())
            finished = true;
        break;
    case POST_ORDER:
        postQue.Dequeue(item);
        if (postQue.IsEmpty())
            finished = true;
        break;
    }
}
```

::::::::::::maincpp::::::::::::

```
#include <iostream>
#include <string>
#include "binarysearchtree.h"
#include "binarysearchtree.cpp"
using namespace std;

struct Passenger
{
    string name;

    bool operator<(const Passenger &other) const
    {
        return name < other.name;
    }
};

struct Flight
{
    string name;
    TreeType<Passenger> passengerList;

    bool operator<(const Flight &other) const
    {
        return name < other.name;
    }

    Flight(string flightName) : name(flightName) {}
};

int main()
{
    TreeType<Flight> listOfFlight;

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

        switch (choice)
        {
        case 1:
        {
            string flightName, passengerName;
            cout << "Enter flight name: ";
            cin >> flightName;
            cout << "Enter passenger name: ";
            cin >> passengerName;
```

```
        Flight flight(flightName);
        Passenger passenger;
        passenger.name = passengerName;

        bool found;
        listOfFlight.RetrieveItem(flight, found);

        if (found)
        {
            flight.passengerList.InsertItem(passenger);
        }
        else
        {
            flight.passengerList.MakeEmpty();
            flight.passengerList.InsertItem(passenger);
            listOfFlight.InsertItem(flight);
        }

        cout << "Ticket reserved successfully!\n";
        break;
    }

    case 2:
    {
        string flightName, passengerName;
        cout << "Enter flight name: ";
        cin >> flightName;
        cout << "Enter passenger name: ";
        cin >> passengerName;

        Flight flight(flightName);
        Passenger passenger;
        passenger.name = passengerName;

        bool found;
        listOfFlight.RetrieveItem(flight, found);

        if (found)
        {
            flight.passengerList.DeleteItem(passenger);
            cout << "Reservation canceled successfully!\n";
        }
        else
        {
            cout << "Flight not found or passenger not on the flight.\n";
        }
        break;
    }

    case 3:
    {
        string passengerName;
        cout << "Enter passenger name: ";
        cin >> passengerName;
```

```
            TreeType<Flight> realFlight;
            bool found;
            Flight tempFlight;

            for (int i = 0; i < listOfFlight.LengthIs(); i++)
            {
                listOfFlight.GetNextItem(tempFlight, IN_ORDER, found);
                Passenger passengerToFind;
                passengerToFind.name = passengerName;

                Passenger passengerFound =
tempFlight.passengerList.RetrieveItem(passengerToFind, found);

                if (found && passengerFound.name != "")
                {
                    realFlight.InsertItem(tempFlight);
                }
            }

            if (realFlight.LengthIs() > 0)
            {
                cout << "Passenger " << passengerName << " is on the following
flights:\n";
                Flight passengerFlight;
                for (int i = 0; i < realFlight.LengthIs(); i++)
                {
                    realFlight.GetNextItem(passengerFlight, IN_ORDER, found);
                    cout << passengerFlight.name << endl;
                }
            }
            else
            {
                cout << "Passenger not found on any flight.\n";
            }
            break;
        }

        case 4:
        {
            string flightName;
            cout << "Enter flight name: ";
            cin >> flightName;

            Flight flight(flightName);
            bool found;
            listOfFlight.RetrieveItem(flight, found);

            if (found)
            {
                cout << "Passengers on flight " << flightName << ":\n";
                Passenger passenger;
                for (int i = 0; i < flight.passengerList.LengthIs(); i++)
                {
                    flight.passengerList.GetNextItem(passenger, IN_ORDER, found);
                    cout << passenger.name << endl;
                }
```

```
            }
            else
            {
                cout << "Flight not found.\n";
            }
            break;
        }

        case 5:
            cout << "Exiting program.\n";
            break;

        default:
            cout << "Invalid choice. Please try again.\n";
            break;
        }
    } while (choice != 5);

    return 0;
}
```