**VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY**

**THE INTERNATIONAL UNIVERSITY**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**------\*\*\*------**



**ALGORITHMS AND DATA STRUCTURES**

**IT013IU**

PROJECT REPORT

**IU SCHEDULER WEBSITE APPLICATION**

**Member List**

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Full name | Student’s ID | Team role |
| 1 | Hoàng Công Gia Huy | ITITDK23038 | Team leader |

Instructor: **Mr. Thai Trung Tin**

**ACKNOWLEGEMENT**

I would like to express my heartfelt gratitude to my instructors, Dr. Vi Chi Thanh and MSc. Thai Trung Tin, for their invaluable guidance, encouragement, and support throughout this project. Their insightful advice and constructive feedback have been instrumental in shaping the development of the Minesweeper game with an undo feature.

While I faced several difficulties in understanding and implementing certain aspects of the project, these challenges proved to be significant learning opportunities. They allowed me to enhance my problem-solving abilities, knowledge about algorithms and data structures and to develop a game with the implementation of algorithms and data structures.

This project has been a valuable learning experience, and it would not have been possible without the support of Dr. Vi Chi Thanh and MSc. Thai Trung Tin.

**TABLE OF CONTENTS**

[**CHAPTER I. INTRODUCTION** 5](#_Toc199989632)

[1.1. Abstract 5](#_Toc199989633)

[1.1.2. Techniques 5](#_Toc199989635)

[1.3. Contribution 6](#_Toc199989636)

[**CHAPTER II. DESIGN & IMPLEMENTATION** 7](#_Toc199989637)

[2.1. Front End 7](#_Toc199989638)

[2.1.1. Main Features 7](#_Toc199989639)

[2.2. Back End 8](#_Toc199989640)

[2.1.1. Data Structures 8](#_Toc199989641)

[**CHAPTER III. FINALE** 15](#_Toc199989642)

[**CHAPTER IV. CONCLUSION** 18](#_Toc199989644)

[4.1. Conclusion 18](#_Toc199989645)

[4.2. Experience 18](#_Toc199989646)

# 

# **CHAPTER I. INTRODUCTION**

## 1.1. Abstract

## This project implements an appointment booking system designed to be efficient and user-friendly. It allows users to securely create accounts, schedule appointments by selecting dates and times from an intuitive calendar interface, and view their booking history as well as deleting appointments. The system effectively manages appointment slots to prevent conflicts and ensures that user and booking data are handled effectively on the backend to deliver a responsive experience. This project demonstrates the development of a practical application of data structures for managing schedules.

## 1.1.2. Techniques

* **Frameworks:**  
  **React (ReactJS):** Framework for creating interactive and reusable user interface components.
* **Development Tools:  
  Visual Studio Code:** IDEs/text editors for coding, debugging, and project management.
* **Languages:** **HTML / CSS / TypeScript:** Used for structuring, designing, and adding interactivity

**Python:** Used for implementing backend logic using data structures

* **Design:   
  Figma:** UI design tool for creating collaborative UI/UX designs.

## 1.3. Contribution

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Full name | Student’s ID | Contribution |
| 1 | Hoàng Công Gia Huy | ITITDK23038 | 100% |

## 

# 

# **CHAPTER II. DESIGN & IMPLEMENTATION**

## 2.1. Front End

### 2.1.1. Main Features

**a. User Registration**

* **Files:** SignUp.tsx, SignIn.tsx

A close up of a computer screen

Description automatically generated

* **Description:**
  + Allows new users to create an account by providing essential details such as their name, email, and a password.
  + The system ensures that each email is unique, preventing duplicate user registrations. This is managed by the backend which checks for existing users in a Hash Table.
  + User accounts and credentials are managed on the backend using a Hash Table data structure, which allows for efficient storage and retrieval of user information.

**b. Booking System**

* **Files:** MainBoard.tsx

A close up of a sign

Description automatically generated

* **Description:**
  + Features an interactive calendar interface, allowing users to visually select a preferred date for their appointment.
  + Provides a selection of available time slots for the chosen date.
  + To prevent scheduling conflicts, the system checks if a selected date and time slot is already booked. This is managed on the backend by a balancing Binary Search Tree which stores appointments for efficient lookup.
  + Once a valid slot is chosen and confirmed, the appointment is recorded.

**c. View Appointment History**

* **Files:** MainBoard.tsx

A close up of a sign

Description automatically generated

* **Description:**
  + Allows authenticated users to view a list of their scheduled appointments.
  + The system retrieves and displays appointments specific to the logged-in user.
  + Appointment history for each user is maintained chronologically using a Doubly Linked List data structure on the backend, facilitating easy traversal and management of booking records.

**d. Back End Integration**

* File: *api.ts*

A blue and white background

Description automatically generated with medium confidence

* Description:
* Integrates with backend services to fetch data.

## 2.2. Back End

### 2.1.1. Data Structures

**a. HashTable**

*-* ***hash(self, key)***

* Time Complexity: O(L). Basic hash() function, its complexity is linear with respect to the key length, L.

**- *insert(self, key, value)***

* Average Case Time Complexity: O(L + α), where α is the bounded load factor.
* Worst Case Time Complexity: O(L + N). This scenario arises if all N keys collide into a single bucket, degenerating the chain into a linear list of length N that must be traversed.
* Space Complexity: O(1)

**- *get(self, key) (Retrieval)***

* Average Case Time Complexity: O(L + α), simplifying to O(1) under ideal hashing conditions and a low load factor.
* Worst Case Time Complexity: O(L + N), analogous to insertion, due to potential full chain traversal.

A screen shot of a computer program

Description automatically generated

**b. Linked List**

- ***add\_appointment(self, date, time, username)***

* Time Complexity: O(1)
  + Creates a new node.
  + Updates self.tail and the next pointer of the old tail, and prev pointer of the new node. All are constant time operations.
* Space Complexity: O(1)

**- *get\_user\_appointments(self, username)***

* Time Complexity: O(N)
  + Ttraverse the entire list from head to tail to find all appointments for a specific user.
* Space Complexity:O(k) where k is the number of appointments for that specific user. In the worst case, if all appointments belong to one user, it's O(N).

**- *remove\_appointment(self, username)***

* Time Complexity: O(N)
  + Iterate through the list to find the first appointment matching the username. This is O(N) in the worst case.
  + Once the node is found, the actual removal is O(1) because it's a doubly linked list.
* Space Complexity: O(1).

A screen shot of a computer program

Description automatically generated

**c. Self-Balancing Binary Search Tree**

* **Insert appointment**

- ***insert\_node(self, node, date, time, username)***

* Time Complexity: O(log N)
  + Traversal: In a balanced BST, traversing takes O(log N)
  + Balancing: After insertion, the tree might become unbalanced. The balancing process involves checking balance factors and performing rotations. At most two rotations are needed for an AVL tree insertion. These rotations and height updates propagate up the tree from the insertion point to the root, taking O(H) = O(log N) time.
* Space Complexity (for recursion stack): O(log N) due to the recursive calls.

A computer screen shot of a code

Description automatically generated

* **Delete appointment**

- ***delete\_node(self, node, date, time, username)***

* Time Complexity: O(log N)
  + Traversal: In a balanced AVL tree, finding the node to delete takes O(log N) time.
  + **Deletion**: Once the node is found:
    - If it has no or one child, removal takes constant time
    - If it has two children, replacing with the inorder successor also takes O(log N) in the worst case.
  + Balancing: After deletion, the tree might become unbalanced. The balancing process involves checking balance factors and performing rotations. At most two rotations are needed for an AVL tree insertion. These rotations and height updates propagate up the tree from the insertion point to the root, taking O(H) = O(log N) time.
* Space Complexity (for recursion stack): O(log N) due to the recursive calls.

A screen shot of a computer code

Description automatically generated

* Check for Imbalance and Perform Rotations:
* **Left-Heavy Cases (**balance > 1**)**:
  + **Left-Left (LL) Case**: *If balance > 1* *and* *self.get\_balance(node.left) >= 0*, a single right rotation on node is performed:  
    *return self.rotate\_right(node)*
  + **Left-Right (LR) Case**: *If balance > 1 and self.get\_balance(node.left) < 0*, a left rotation on node.left is performed first (*node.left = self.rotate\_left(node.left)*), followed by a right rotation on node:  
    *return self.rotate\_right(node)*
* **Right-Heavy Cases (**balance < -1**)**:
  + **Right-Right (RR) Case***: If balance < -1 and self.get\_balance(node.right) <= 0*, a single left rotation on node is performed:  
    *return self.rotate\_left(node)*
  + **Right-Left (RL) Case***: If balance < -1 and self.get\_balance(node.right) > 0*, a right rotation on node.right is performed first *(node.right = self.rotate\_right(node.right)*), followed by a left rotation on node:  
    *return self.rotate\_left(node)*

A screen shot of a computer program

Description automatically generated

# **CHAPTER III. FINALE**

A screenshot of a calendar

Description automatically generated

**Figure 3.1**: Home page

A screenshot of a computer

Description automatically generated

**Figure 3.2**: Sign-up

# **A screenshot of a computer Description automatically generated**

**Figure 3.3**: Sign-in

A white rectangular object with blue lines

Description automatically generated

**Figure 3.4**: Booking History

# 

# **CHAPTER IV. CONCLUSION**

## 4.1. Conclusion

This project successfully developed an appointment booking simulation by implementing key data structures for an efficient system. A Hash Table was used for O(1) average-time user authentication, an AVL Tree ensured O(log n) appointment conflict checking and sorted management, and a Doubly Linked List enabled O(1) booking history insertions. These strategically chosen data structures proved instrumental in straightforwardly scheduling and tracking appointments. The project highlights the crucial role of appropriate DSA selection in building responsive, scalable, and good performance application.

## 4.2. Experience

Working on this booking tool was a great way to learn about data structures. I built a Hash Table for user accounts, an AVL Tree for appointments, and a Doubly Linked List for booking history. This involved hands-on work with hashing, tree balancing, and how data connects. It showed me why these structures are vital for speed, like for quick user logins, fast appointment searches, and adding to history without delay. Putting it all together and fixing issues by myself taught me to plan carefully and always improve. This project boosted my skills in using data structures to create efficient software and helped me understand how they make everyday tasks simpler for users.

# 