PROJECT TITLE:
EMPLOYEE MANAGEMENT SYSTEM

GROUP MEMBERS:

NO	NAME	MATRIC NO
1.	ARUN KUMAR A/L MANIMARAN	CI230128
2.	CHE AZFAR SYAKIRIN BIN CIK ANUAR	CI230095
3.	MASIDAH BINTI SINTO	CI230051
4.	KHOR HWEE SHIAN	CI230085
5.	AKMAL HAKIMI BIN ADNAN	CI230065

UNIVERSITI TUN HUSSEIN ONN MALAYSIA

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Design issues

There are several design issues in the manual employee management system compared to the system we have created. Firstly, the manual employee system consumes more time and occur many errors when using the manual employee management system. Physical documents are susceptible to loss, damage, and degradation over time. By using this system, the data entry process becomes more user friendly and can be easily used by the users. This system also utilizes digital storage solutions to maintain and protect data, reducing the risk of physical damage or loss. Next, manual records make it difficult to access the records we need quickly. Physical documents are vulnerable to unauthorized access and theft. We overcome this problem using centralized storing in this system.

Manual process makes it difficult to create accurate reports using the data in the manual employee management system. In our employee management system, all the data is arranged orderly, which makes the report generation easy. Other than that, in the manual system employees must rely on HR for basic information and updates, leading to inefficiencies. Lack of self-service options can decrease employee satisfaction and engagement. We solve this issue by implementing a user-friendly interface that can enable the user to update their information. Moreover, Manual systems struggle to scale with organizational growth. We solve this issue by designing an employee management system with scalable architecture to accommodate growth.

1.2 Problems with Manual Employee Management

Managing employee records manually can lead to several significant challenges, which can affect the overall efficiency and accuracy of an organization's operations. One of the primary issues is the potential for human error. When records are manually entered, updated, and maintained on paper or in basic digital formats, there is a high risk of typographical errors, data duplication, and loss of information. For example, an incorrect entry of an employee's NRIC or salary due to a simple keystroke error can cause substantial discrepancies in payroll processing and legal compliance.

Additionally, manual management systems are inherently inefficient when it comes to data retrieval and reporting. Finding specific information about an employee or generating reports can be time-consuming and cumbersome. For instance, if a manager needs to retrieve all the records of employees from a particular department, they might have to sift through numerous files or documents. This inefficiency can lead to delays in decision-making and an inability to quickly respond to business needs or audits. Moreover, the lack of a centralized database makes it difficult to ensure data consistency and integrity, as changes in one document might not be reflected in another.

Security and confidentiality are also significant concerns in manual management systems. Physical records can be easily misplaced, stolen, or damaged by accidents such as fires or floods. Moreover, unauthorized access to sensitive employee information is harder to control without proper access management systems in place. A breach of employee records can lead to identity theft, financial fraud, and legal ramifications for the company. Therefore, transitioning to an automated system, such as the C program described, addresses these problems by providing robust validation, efficient data retrieval, secure storage, and easy scalability.

1.3 Solutions with Computerized Systems

Computerized employee management with a structured C program, as described, effectively mitigates these problems. First, the program's validation functions (isValidName and isValidNRIC) significantly reduce the risk of human error by ensuring that only correctly formatted data is entered into the system. This improves the accuracy of the records and ensures consistency across the database. Additionally, by using a digital system, data can be easily backed up and restored, protecting against data loss due to physical damage or human error.

The efficiency of data retrieval and reporting is vastly improved with a computerized system. Functions such as display data employees by sorting in order, search data employee, and update details of employees allow for quick and easy access to specific employee information and the generation of reports. For example, a manager can instantly retrieve all employees in a department or sort employees by employee’s ID. This immediacy facilitates better decision-making and responsiveness to business needs. Moreover, the program can handle large volumes of data without the performance degradation that a manual system would experience.

1.4 Data Structure (Reasons behind selecting them)

The linked list data structure offers several advantages for managing dynamic and variable-sized data, making it a suitable choice for the employee management system described. Its ability to handle dynamic memory allocation efficiently, ease of insertion and deletion operations, and flexibility in data organization make it an ideal solution for this application.

The C program for managing employee records uses a linked list data structure to store and manage employee information. The primary reasons for selecting a linked list over other data structures are as follows. Firstly, dynamic memory allocation by using linked lists allows for dynamic memory allocation, meaning the program can efficiently manage memory usage as it only allocates memory when an employee is added. Unlike arrays, where a predefined size must be set, linked lists can grow and shrink in size as needed. This is particularly useful for applications where the number of employees can vary significantly over time. In this program, each `Employee` node is created dynamically using `malloc`, ensuring that memory is allocated only when needed.

Linked lists provide a flexible and dynamic data structure for managing employee records. It can easily grow and shrink in size as employees are added and removed. There is no need to define a maximum number of employees ahead of time. Memory allocation for linked lists is done dynamically. Each node is created only when needed, which can lead to more efficient memory usage. Finally, Linked lists simplify the implementation of more complex data structures, such as stacks, queues, and other variations like doubly linked lists or circular linked lists.

In the context of an employee management system, arrays offer a simple yet effective way to store and retrieve employee data quickly. Arrays are simple to implement and use, which can make them an attractive choice for straightforward tasks like storing and retrieving employee records. Arrays allocate memory contiguously, which can be more memory-efficient and improve cache performance compared to non-contiguous data structures.

Besides that, ease of insertion and deletion records in a linked list is straightforward and efficient. In an array, inserting or deleting an element requires shifting all subsequent elements, which can be time-consuming. However, in a linked list, these operations can be performed by simply updating the pointers. This is beneficial for

operations such as adding a new employee or deleting an existing one, where the program needs to quickly adjust the list without extensive memory operations.

Structs (structures) in C programming allow you to group related data together. For instance, an employee's information such as ID, name, department, and salary can be encapsulated within a single struct, making the data more organized and easier to manage. Structs enforce strong typing, which helps prevent errors related to incorrect data manipulation. Each field in a struct has a specific data type, ensuring that only appropriate data is assigned to each field. Finally, struct typically have a contiguous memory layout, which can be more cache-friendly and efficient in terms of memory access compared to classes or dynamic data structures.

Validation is a critical component of the employee management system, ensuring that the data entered into the system is consistent with predefined formats and standards. For example, validating that names contain only alphabetic characters prevent the entry of invalid characters. It can reduce the number of errors that can be made by the users when using the system. It makes sure the data that is entered is accurate and makes sense. Validation inputs to prevent malicious inputs, such as SQL injection or other forms of attacks.

1.5 Algorithms

The Employee Management System employs several algorithms for different functions, each tailored to meet specific requirements efficiently. Here’s an analysis and justification of the algorithms used; firstly, when adding new employee details, the system allocates memory for a new employee node. The user is prompted to enter the employee's ID, name, NRIC, department, and salary. This input is stored in the newly allocated node. The new node is then inserted at the beginning of the linked list by setting its next pointer to the current head of the list and updating the head to point to this new node. This method

is chosen for its simplicity and efficiency, with a time complexity of O(1) for insertion, making it an effective way to manage dynamic data.

Besides that, updating employee details involves first displaying all employees to help the user select the correct record. The user is then prompted to enter the employee ID of the record they wish to update. The system traverses the linked list to find the node with the matching ID. Once found, the user is prompted for new details, which are then updated in the node. This approach ensures that the correct record is modified and leverages the straightforward traversal and update operations of linked lists, albeit with a linear time complexity of O(n) for search.

To delete an employee record, the user is prompted for the employee ID to be deleted. The system then traverses the linked list to find the node with the matching ID. Once identified, the node is removed from the linked list, and its memory is freed. If the node is the head, the head is updated to the next node; otherwise, the previous node's next pointer is updated to skip the deleted node. This operation efficiently removes nodes with a time complexity of O(n), leveraging the dynamic nature of linked lists to handle deletions gracefully.

Displaying employee details involves traversing the linked list from the head to the end and printing the details of each employee node. This linear traversal ensures that all employee records are displayed in the order they are stored, providing a clear and straightforward output for the user. The simplicity of this approach is well-suited for the dynamic data structure of linked lists, with a time complexity of O(n).

Searching for an employee involves prompting the user for the employee ID to be searched. The system then traverses the linked list to find the node with the matching ID. If found, the details of the employee are printed. This linear search, with a time complexity of O(n), ensures that the user can locate specific records efficiently, taking advantage of

the linked list structure's straightforward traversal. Other than that, saving employee details to a file is achieved by opening a file in binary write mode and traversing the linked list to write each employee node to the file. After all nodes are written, the file is closed. This process ensures that all employee data is persistently stored, allowing for recovery and continuity of data across sessions. The time complexity is O(n), making it efficient for moderate-sized datasets.

The main menu selection algorithm displays the menu options and prompts the user for their choice. Based on the user's input, the corresponding function is called. This loop continues until the user chooses to save and exit. This interactive approach ensures that users can seamlessly navigate and utilize the system's functionalities, with each operation efficiently managed through appropriate algorithms tailored to the dynamic nature of linked lists. In summary, the system employs simple yet effective algorithms to manage employee records efficiently. While more advanced data structures and algorithms could offer improved performance for larger datasets, the current implementation strikes a balance between simplicity, efficiency, and ease of use for the intended scale.

1.6 Emphasis of Algorithms

In a linked list, adding employee details is a highly efficient operation with a time complexity. This efficiency stems from the process of adding a new node at the head of the list, which involves only a few pointer updates. The new employee is added to the head of the linked list, which is a constant-time operation. These steps are constant-time operations, meaning the time required does not increase with the size of the list. As a result, adding an employee is very fast and does not depend on the number of nodes already present in the list.

Conversely, operations such as updating, deleting, and searching for employee details have a time complexity. This is because these operations may require traversing the list to locate the specific node containing the employee's information. Since a linked list does not allow direct access to individual nodes, each node must be checked sequentially, leading to a linear time complexity. In the worst case, the algorithm might need to traverse the entire list, making these operations slower as the list grows.

Displaying all employee details also has a time complexity as it requires traversing the entire list to access and print each node's data. This linear complexity indicates that the time taken to display the list increases proportionally with the number of nodes. While the linked list structure allows for efficient insertion at the head, the need for traversal in other operations highlights the trade-offs in its use, particularly as the number of employees increases.

1.7 Optimization Issues

To further optimize the algorithms and address the mentioned issues, several strategies can be employed: Firstly, for the sorting algorithm optimization using QuickSort, can be used to avoid future complexities. It’s beneficial to implement adaptive strategies that address both average and worst-case scenarios. This includes incorporating algorithms like Insertion Sort for small arrays or Merge Sort for larger arrays when QuickSort's performance might degrade due to poor pivot selection or near-sorted data. By adopting adaptive sorting techniques, the algorithm becomes more versatile and efficient across a wide range of input sizes and distributions, ensuring consistent performance in real-world applications.

Besides that, optimized file I/O operations involves adding hash table algorithm to make the search algorithm more efficient. Techniques such as prefetching, compression,

and optimizing access patterns can significantly enhance performance. Prefetching data into memory ahead of time reduces latency by anticipating future accesses, while compression minimizes the amount of data read or written, thereby speeding up I/O operations. Lastly, using more descriptive variable names by improving code readability and maintainability by using more descriptive names.

CHAPTER 2

SYSTEM EXPLANATION

2.1 Screenshot of Coding

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Structure to represent an Employee
typedef struct Employee {
    int id;
    char name[50];
    char nric[20];
    char department[50];
    float salary;
    struct Employee* next;
} Employee;

Employee* head = NULL;

void addEmployee();
void updateEmployee();
void deleteEmployee();
void displayEmployees();
void searchEmployee();
void sortEmployees();
void menu();
void saveEmployeesToFile();
void loadEmployeesFromFile();

// Function to check if a name is valid
```

```c
int isValidName(char* name) {
    for (int i = 0; name[i] != '\0'; i++) {
        if (!isalpha(name[i]) && name[i] != ' ') {
            return 0;
        }
    }
    return 1;
}

// Function to check if an NRIC is valid
int isValidNRIC(char* nric) {
    for (int i = 0; nric[i] != '\0'; i++) {
        if (!isdigit(nric[i])) {
            return 0;
        }
    }
    return 1;
}

// Function to add a new employee
void addEmployee() {
    // Allocate memory for a new employee
    Employee* newEmployee = (Employee*)malloc(sizeof(Employee));
    if (!newEmployee) {
        // Display error message if memory allocation fails
        printf("\033[91mCannot add more employees. Memory allocation failed.
[!]\033[0m\n");
        return;
    }

    printf("\033[1mEnter Employee ID: \033[0m");
    while (scanf("%d", &newEmployee->id) != 1) {
        // Handle invalid input for ID
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee ID: \033[0m");
        while (getchar() != '\n');
    }

    getchar();

    // Get input for Employee Name
    printf("\033[1mEnter Employee Name: \033[0m");
    fgets(newEmployee->name, sizeof(newEmployee->name), stdin);
    newEmployee->name[strcspn(newEmployee->name, "\n")] = 0;
    while (!isValidName(newEmployee->name)) {
```

```
        // Handle invalid input for Name
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee ID: \033[0m");
        fgets(newEmployee->name, sizeof(newEmployee->name), stdin);
        newEmployee->name[strcspn(newEmployee->name, "\n")] = 0;
    }

    // Get input for Employee NRIC
    printf("\033[1mEnter Employee NRIC: \033[0m");
    fgets(newEmployee->nric, sizeof(newEmployee->nric), stdin);
    newEmployee->nric[strcspn(newEmployee->nric, "\n")] = 0;
    while (!isValidNRIC(newEmployee->nric)) {
        // Handle invalid input for NRIC
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee NRIC:  \033[0m");
        fgets(newEmployee->nric, sizeof(newEmployee->nric), stdin);
        newEmployee->nric[strcspn(newEmployee->nric, "\n")] = 0;
    }

    // Get input for Employee Department
    printf("\033[1mEnter Employee Department: \033[0m");
    fgets(newEmployee->department, sizeof(newEmployee->department), stdin);
    newEmployee->department[strcspn(newEmployee->department, "\n")] = 0;

    // Get input for Employee Salary
    printf("\033[1mEnter Employee Salary: \033[0m");
    while (scanf("%f", &newEmployee->salary) != 1) {
        // Handle invalid input for Salary
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Salary: \033[0m");
        while (getchar() != '\n');
    }

    // Insert the new employee into the linked list
    newEmployee->next = head;
    head = newEmployee;

    printf("\033[92mEmployee with ID %d added successfully!\033[0m\n",
newEmployee->id);
}

// Function to update employee details
void updateEmployee() {
    if (head == NULL) {
        printf("\033[92mNo employees to update. [!]\033[0m\n");
```

```
        return;
    }
    displayEmployees();

    int id;
    printf("\033[1mEnter Employee ID to update: \033[0m");
    while (scanf("%d", &id) != 1) {
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee ID: \033[0m");
        while (getchar() != '\n');
    }

    Employee* temp = head;
    while (temp != NULL) {
        if (temp->id == id) {
            getchar();

            printf("\033[94mChoose the field to update:\n");
            printf("1. Name\n");
            printf("2. NRIC\n");
            printf("3. Department\n");
            printf("4. Salary \033[0m\n");
            printf("\033[1mEnter your choice: \033[0m");
            int choice;
            while (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
                printf("\033[91mInvalid input [!] \033[0m\n");
                printf("\033[1mEnter a valid choice (1-4): \033[0m");
                while (getchar() != '\n');
            }

            getchar();

            switch (choice) {
                case 1:
                    printf("\033[1mEnter new Employee Name: \033[0m");
                    fgets(temp->name, sizeof(temp->name), stdin);
                    temp->name[strcspn(temp->name, "\n")] = 0;
                    while (!isValidName(temp->name)) {
                        printf("\033[91mInvalid input [!] \033[0m\n");
                        printf("\033[1mEnter a valid Employee Name: \033[0m");
                        fgets(temp->name, sizeof(temp->name), stdin);
                        temp->name[strcspn(temp->name, "\n")] = 0;
                    }
                    break;
                case 2:
```

```
                    printf("\033[1mEnter new Employee NRIC (without dash):
\033[0m");
                    fgets(temp->nric, sizeof(temp->nric), stdin);
                    temp->nric[strcspn(temp->nric, "\n")] = 0;
                    while (!isValidNRIC(temp->nric)) {
                        printf("\033[91mInvalid input [!] \033[0m\n");
                        printf("\033[1mEnter a valid Employee NRIC: \033[0m");
                        fgets(temp->nric, sizeof(temp->nric), stdin);
                        temp->nric[strcspn(temp->nric, "\n")] = 0;
                    }
                    break;
                case 3:
                    printf("\033[1mEnter new Employee Department: \033[0m");
                    fgets(temp->department, sizeof(temp->department), stdin);
                    temp->department[strcspn(temp->department, "\n")] = 0;
                    break;
                case 4:
                    printf("\033[1mEnter new Employee Salary: \033[0m");
                    while (scanf("%f", &temp->salary) != 1) {
                        printf("\033[91mInvalid input [!] \033[0m\n");
                        printf("\033[1mEnter a valid Salary: \033[0m");
                        while (getchar() != '\n');
                    }
                    break;
            }

            printf("\033[92mEmployee with ID %d updated successfully!\033[0m\n",
id);
            return;
        }
        temp = temp->next;
    }

    printf("\033[91mEmployee with ID %d not found.\033[0m\n", id);
}

// Function to delete an employee
void deleteEmployee() {
    if (head == NULL) {
        printf("\033[92mNo employees to delete. [!]\033[0m\n");
        return;
    }

    int id;
    displayEmployees();
```

```
    printf("\033[1mEnter Employee ID to delete: \033[0m");
    while (scanf("%d", &id) != 1) {
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee ID: \033[0m");
        while (getchar() != '\n');
    }

    Employee* temp = head;
    Employee* prev = NULL;

    if (temp != NULL && temp->id == id) {
        head = temp->next;
        free(temp);
        printf("\033[92mEmployee with ID %d deleted successfully!\033[0m\n", id);
        return;
    }

    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("\033[91mEmployee with ID %d not found.\033[0m\n", id);
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("\033[92mEmployee with ID %d deleted successfully!\033[0m\n", id);
}

// Function to sort employees by ID
void sortEmployees() {
    if (head == NULL) return;

    Employee* sorted = NULL;
    Employee* current = head;

    while (current != NULL) {
        Employee* next = current->next;
        if (sorted == NULL || sorted->id >= current->id) {
            current->next = sorted;
            sorted = current;
        } else {
```

```
            Employee* temp = sorted;
            while (temp->next != NULL && temp->next->id < current->id) {
                temp = temp->next;
            }
            current->next = temp->next;
            temp->next = current;
        }
        current = next;
    }
    head = sorted;
}

// Function to display all employees
void displayEmployees() {
    sortEmployees();
    Employee* temp = head;

    if (temp == NULL) {
        printf("\033[91mNo employees to display. [!]\033[0m\n");
        return;
    }

    printf("\033[94m-------------------------------------------------------------
------------------------------\n");
    printf("| %-10s | %-20s | %-15s | %-20s | %-10s |\n", "ID", "Name", "NRIC",
"Department", "Salary");
    printf("-------------------------------------------------------------------
----------------------\n");

    while (temp != NULL) {
        printf("| %-10d | %-20s | %-15s | %-20s | %-10.2f |\n",
               temp->id, temp->name, temp->nric, temp->department, temp->salary);
        temp = temp->next;
    }

    printf("-------------------------------------------------------------------
----------------------\n\033[0m");
}

// Function to find an employee by ID
Employee* findEmployeeById(int id) {
    Employee* temp = head;
    while (temp != NULL) {
        if (temp->id == id) { // Check if the current employee's ID matches the
input ID
```

```
            return temp;
        }
        temp = temp->next; // Move to the next employee in the list
    }
    return NULL;
}

// Function to print employee details
void printEmployee(Employee* emp) {
    if (!emp) {// Check if the employee data is available
        printf("\033[91mEmployee data is not available. [!]\033[0m\n");
        return;
    }
     // Print employee details
    printf("\033[94m-----------------------------------------------------\n");
    printf("| %-15s | %-30s |\n", "Employee", "Data");
    printf("-----------------------------------------------------\n");
    printf("| %-15s | %-30d |\n", "ID", emp->id);
    printf("| %-15s | %-30s |\n", "Name", emp->name);
    printf("| %-15s | %-30s |\n", "NRIC", emp->nric);
    printf("| %-15s | %-30s |\n", "Department", emp->department);
    printf("| %-15s | %-30.2f |\n", "Salary", emp->salary);
    printf("-----------------------------------------------------\n\033[0m");
}

// Function to search for an employee
void searchEmployee() {
    int id;
    printf("\033[1mEnter Employee ID to search: \033[0m");
    while (scanf("%d", &id) != 1) {
        printf("\033[91mInvalid input [!] \033[0m\n");
        printf("\033[1mEnter a valid Employee ID: \033[0m");
        while (getchar() != '\n');
    }

    Employee* emp = findEmployeeById(id);
    if (emp) {
        printEmployee(emp);
    } else {
        printf("\033[91mEmployee with ID %d not found.\033[0m\n", id);
    }
}

// Function to save employees to a file
void saveEmployeesToFile() {
```

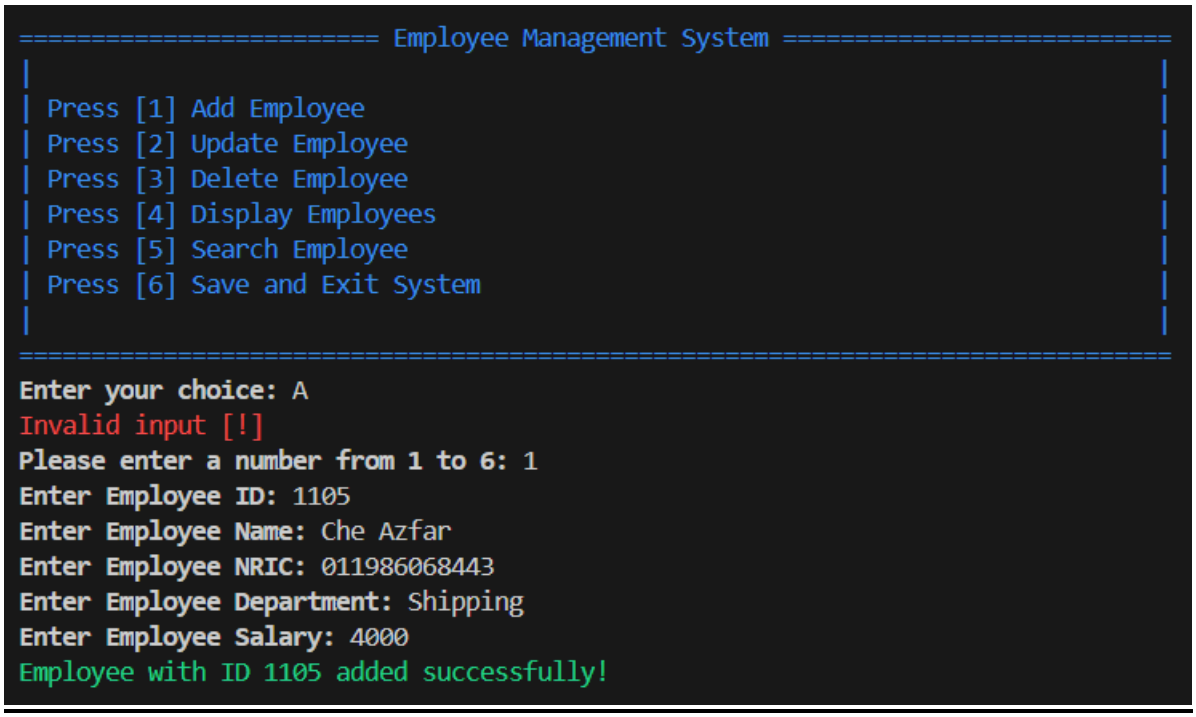
```
    FILE* file = fopen("employees.dat", "wb");
    if (file == NULL) {
        printf("\033[91mError opening file for saving [!]\033[0m\n");
        return;
    }

    Employee* temp = head;
    while (temp != NULL) {
        if (fwrite(temp, sizeof(Employee), 1, file) != 1) {
            printf("\033[91mError writing to file [!]\033[0m\n");
            fclose(file);
            return;
        }
        temp = temp->next;
    }
    fclose(file);
}

// Function to load employees from a file
void loadEmployeesFromFile() {
    FILE* file = fopen("employees.dat", "rb");
    if (file == NULL) {
        printf("\033[91mNo saved data found [!]\033[0m\n");
        return;
    }

    Employee* temp = NULL;
    while (1) {
        Employee* newEmployee = (Employee*)malloc(sizeof(Employee));
        if (!newEmployee) {
            printf("\033[91mMemory allocation failed [!]\033[0m\n");
            fclose(file);
            return;
        }
        if (fread(newEmployee, sizeof(Employee), 1, file) != 1) {
            free(newEmployee);
            break;
        }
        newEmployee->next = temp;
        temp = newEmployee;
    }
    head = temp;
    fclose(file);
}
```

```c
// Function to display the main menu
void menu() {
    int choice;
    while (1) {
        printf("\033[94m\n======================== Employee Management System
==========================\n");
        printf("|
             |\n");
        printf("| Press [1] Add
Employee                                                    |\n");
        printf("| Press [2] Update
Employee                                                   |\n");
        printf("| Press [3] Delete
Employee                                                   |\n");
        printf("| Press [4] Display
Employees                                                 |\n");
        printf("| Press [5] Search
Employee                                                   |\n");
        printf("| Press [6] Save and Exit
System                                                |\n");
        printf("|
             |\n");
        printf("=================================================================
===============\033[0m\n");
        printf("\033[1mEnter your choice: \033[0m");
        while (scanf("%d", &choice) != 1) {
            printf("\033[91mInvalid input [!] \033[0m\n");
            printf("\033[1mPlease enter a number from 1 to 6: \033[0m");
            while (getchar() != '\n');
        }

        switch (choice) {
            case 1:
                addEmployee();
                break;
            case 2:
                updateEmployee();
                break;
            case 3:
                deleteEmployee();
                break;
            case 4:
                displayEmployees();
                break;
            case 5:
```

```
                searchEmployee();
                break;
            case 6:
                saveEmployeesToFile();
                printf("\033[92mData saved. Exiting the system.\033[0m\n");
                exit(0);
            default:
                printf("\033[91mInvalid input [!] \033[0m\n");
                printf("\033[1mPlease enter a number from 1 to 6: \033[0m");
        }
    }
}

int main() {
    loadEmployeesFromFile();
    menu();
    return 0;
}
```

2.2 Screenshot of Output (With Explanation)

SCREENSHOT OF OUTPUT (ADD EMPLOYEE)



This is the add employee menu of the Employee Management System. Here, you can enter details such as employee ID, employee name, employee NRIC, employee department, and employee salary for new employees. When a new employee is added, a new node is created and inserted at the beginning of the linked list. This operation is efficient as it only requires updating the next pointer of the new node to point to the current head of the list and then updating the head to point to the new node.

SCREENSHOT OF OUTPUT (UPDATE EMPLOYEE)

```
======================== Employee Management System ==========================
|                                                                             |
| Press [1] Add Employee                                                      |
| Press [2] Update Employee                                                   |
| Press [3] Delete Employee                                                   |
| Press [4] Display Employees                                                 |
| Press [5] Search Employee                                                   |
| Press [6] Save and Exit System                                              |
|                                                                             |
===============================================================================
Enter your choice: 2
-------------------------------------------------------------------------------------------
| ID         | Name                | NRIC            | Department           | Salary     |
-------------------------------------------------------------------------------------------
| 1101       | Ida Sinto           | 021230011198    | Human Resource       | 2200.00    |
| 1102       | Mohd Qaziq          | 010507039287    | Human Resource       | 2200.00    |
| 1103       | Aini Sinto          | 981218013766    | Shipping             | 2500.00    |
| 1104       | Aiman               | 020604019832    | Shipping             | 2230.50    |
| 1105       | Che Azfar           | 011986068443    | Shipping             | 4000.00    |
-------------------------------------------------------------------------------------------
Enter Employee ID to update: 1105
Choose the field to update:
1. Name
2. NRIC
3. Department
4. Salary
Enter your choice: 1
Enter new Employee Name: Azfar
Employee with ID 1105 updated successfully!
```

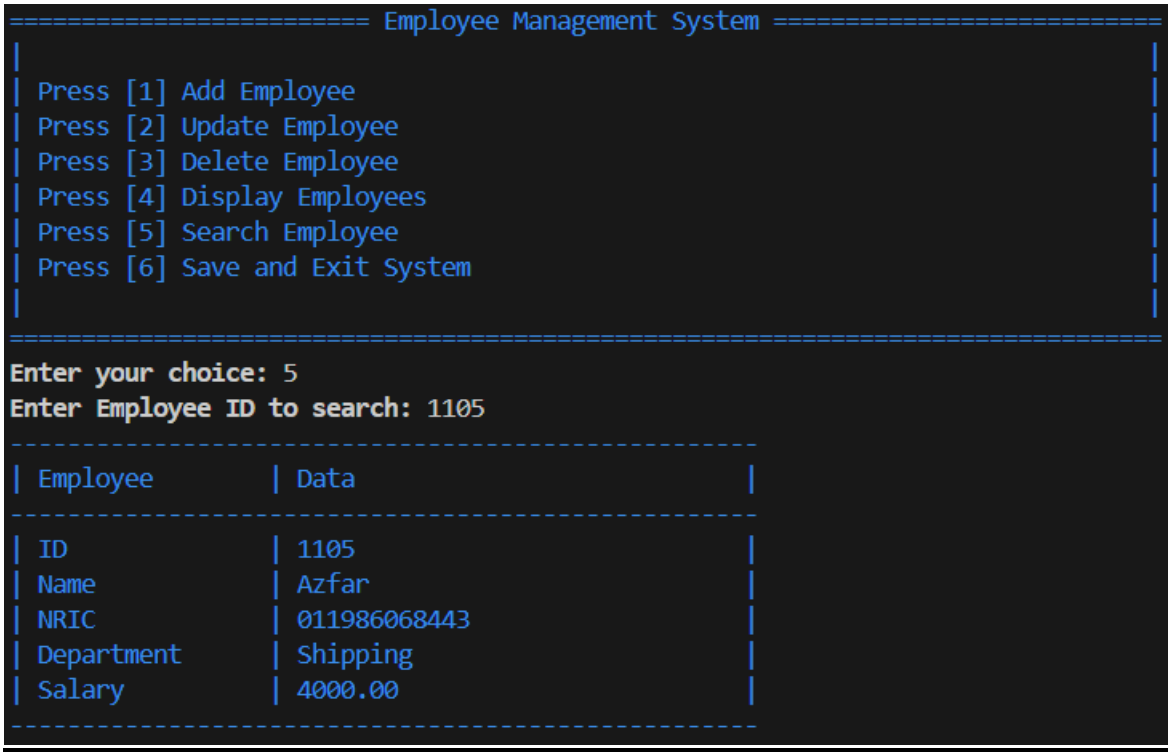
This is the update employee menu of the system. Here, you can update details such as employee name, employee NRIC, employee department, and employee salary for a new employee based on their employee ID. To update an employee's information, the program traverses the linked list to find the node with the matching ID. Once found, it allows the user to update specific fields of the employee record.

SCREENSHOT OF OUTPUT (DISPLAY EMPLOYEE)

```
======================== Employee Management System ============================
|                                                                              |
| Press [1] Add Employee                                                       |
| Press [2] Update Employee                                                    |
| Press [3] Delete Employee                                                    |
| Press [4] Display Employees                                                  |
| Press [5] Search Employee                                                    |
| Press [6] Save and Exit System                                               |
|                                                                              |
================================================================================
Enter your choice: 5
-------------------------------------------------------------------------------------------
| ID         | Name                | NRIC           | Department           | Salary     |
-------------------------------------------------------------------------------------------
| 1101       | Ida Sinto           | 021230011198   | Human Resource       | 2200.00    |
| 1102       | Mohd Qaziq          | 010507039287   | Human Resource       | 2200.00    |
| 1103       | Aini Sinto          | 981218013766   | Shipping             | 2500.00    |
| 1104       | Aiman               | 020604019832   | Shipping             | 2230.50    |
| 1105       | Azfar               | 011986068443   | Shipping             | 4000.00    |
-------------------------------------------------------------------------------------------
Enter Employee ID to search: 1105
-----------------------------------------------
| Employee        | Data                      |
-----------------------------------------------
| ID              | 1105                      |
| Name            | Azfar                     |
| NRIC            | 011986068443              |
| Department      | Shipping                  |
| Salary          | 4000.00                   |
-----------------------------------------------
```

This is the display employee menu of the system. Here, you can view details such as employee ID, employee name, employee NRIC, employee department, and employee salary for new employees. Traversing the linked list to display all employees involves iterating through each node and printing its contents. This process is straightforward due to the simple structure of the linked list.

SCREENSHOT OF OUTPUT (SEARCH EMPLOYEE



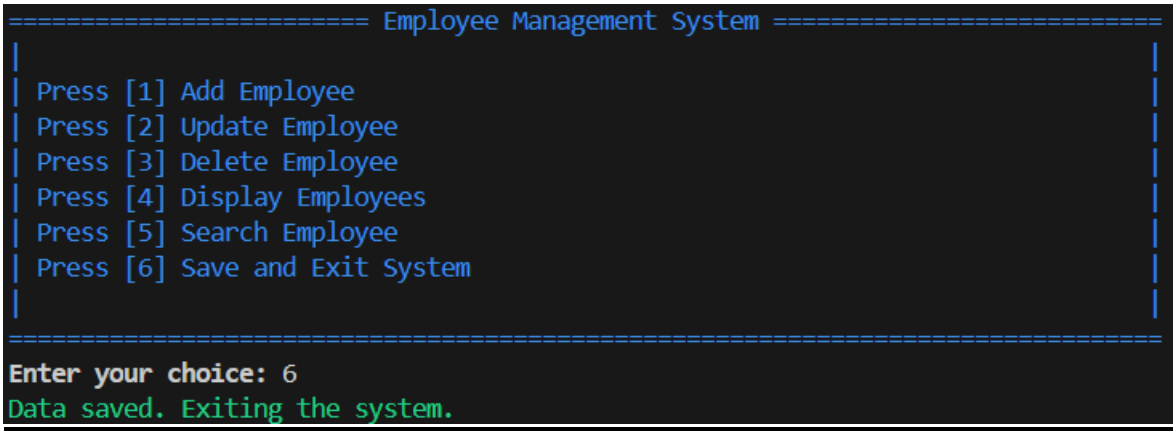
This is the search employee menu of the system. Here, you can search for details such as employee ID, employee name, employee NRIC, employee department, and employee salary for a new employee based on their employee ID, and the system will display the output.

SCREENSHOT OF OUTPUT (DELETE EMPLOYEE)

```
========================= Employee Management System ============================
|                                                                               |
| Press [1] Add Employee                                                        |
| Press [2] Update Employee                                                     |
| Press [3] Delete Employee                                                     |
| Press [4] Display Employees                                                   |
| Press [5] Search Employee                                                     |
| Press [6] Save and Exit System                                                |
|                                                                               |
================================================================================
Enter your choice: 3
-------------------------------------------------------------------------------------------
| ID         | Name                | NRIC           | Department           | Salary     |
-------------------------------------------------------------------------------------------
| 1101       | Ida Sinto           | 021230011198   | Human Resource       | 2200.00    |
| 1102       | Mohd Qaziq          | 010507039287   | Human Resource       | 2200.00    |
| 1103       | Aini Sinto          | 981218013766   | Shipping             | 2500.00    |
| 1104       | Aiman               | 020604019832   | Shipping             | 2230.50    |
| 1105       | Azfar               | 011986068443   | Shipping             | 4000.00    |
-------------------------------------------------------------------------------------------
Enter Employee ID to delete: 1105
Employee with ID 1105 deleted successfully!
```

This is the delete employee menu of the system. Here, you can delete employee details such as employee ID, employee name, employee NRIC, employee department, and employee salary for a new employee based on their employee ID. Although not fully shown in the provided code, deleting an employee involves finding the node with the matching ID and updating the pointers of the adjacent nodes to bypass the node being deleted. This operation can be efficiently performed in a linked list without the need for shifting elements.

SCREENSHOT OF OUTPUT (SAVE AND EXIT THE SYSTEM)



This is the save and exit menu of the system. From here, you can save employee details to a notepad file named employee.dat.

REFERENCES

1. W3Schools.com. (n.d.). https://www.w3schools.com/c/c_functions.php
2. Table Program in C - javatpoint. (n.d.). www.javatpoint.com. https://www.javatpoint.com/table-program-in-c
3. GeeksforGeeks. (2024, February 16). Understanding the basics of Linked List. GeeksforGeeks. https://www.geeksforgeeks.org/what-is-linked-list/
4. *W3Schools.com*. (n.d.-b). https://www.w3schools.com/c/c_structs.php
5. *W3Schools.com*. (n.d.-c). https://www.w3schools.com/c/c_switch.php
6. *W3Schools.com*. (n.d.-d). https://www.w3schools.com/c/c_arrays.php
7. *W3Schools.com*. (n.d.-e). https://www.w3schools.com/c/c_user_input.php