## **Internship Report**

## **C Programming Case Studies**

**Internship Report**  
**C Programming Case Studies**  
**[Your Name]**  
**[Your Internship Organization]**  
**[Date]**

### ****Content****

1. **Abstract**
2. **Introduction**
3. **Case Studies**
   1. Library Management System
   2. Student Database Management System
   3. Banking System
   4. Inventory Management System
   5. Hospital Management System
   6. Employee Payroll System
   7. Tic-Tac-Toe Game
   8. Sorting and Searching Algorithms
4. **Conclusion**
5. **References**

**Abstract**

The objective of this internship report is to present a detailed exploration of various C programming case studies. This report encompasses the design, development, and implementation of ten different projects, each addressing real-world problems through C programming. The case studies include systems for library management, student databases, banking, inventory, hospital management, payroll, games, algorithms . Each case study outlines the problem statement, algorithm, flowchart, implementation details, and screenshots. The report concludes with an overview of the learning outcomes and insights gained throughout the internship.

**Introduction**

The internship focused on developing practical skills in C programming by working on various real-world case studies. Each project was designed to solve specific problems using C, demonstrating core programming concepts and techniques. This report documents the approach taken for each project, including problem statements, algorithms, flowcharts, implementation details, and user interfaces.

**Case Studies**

**CASE- 1: Library Management System**

**Problem Statement:** Design a system to manage books in a library, including functions to add, delete, and search for books.

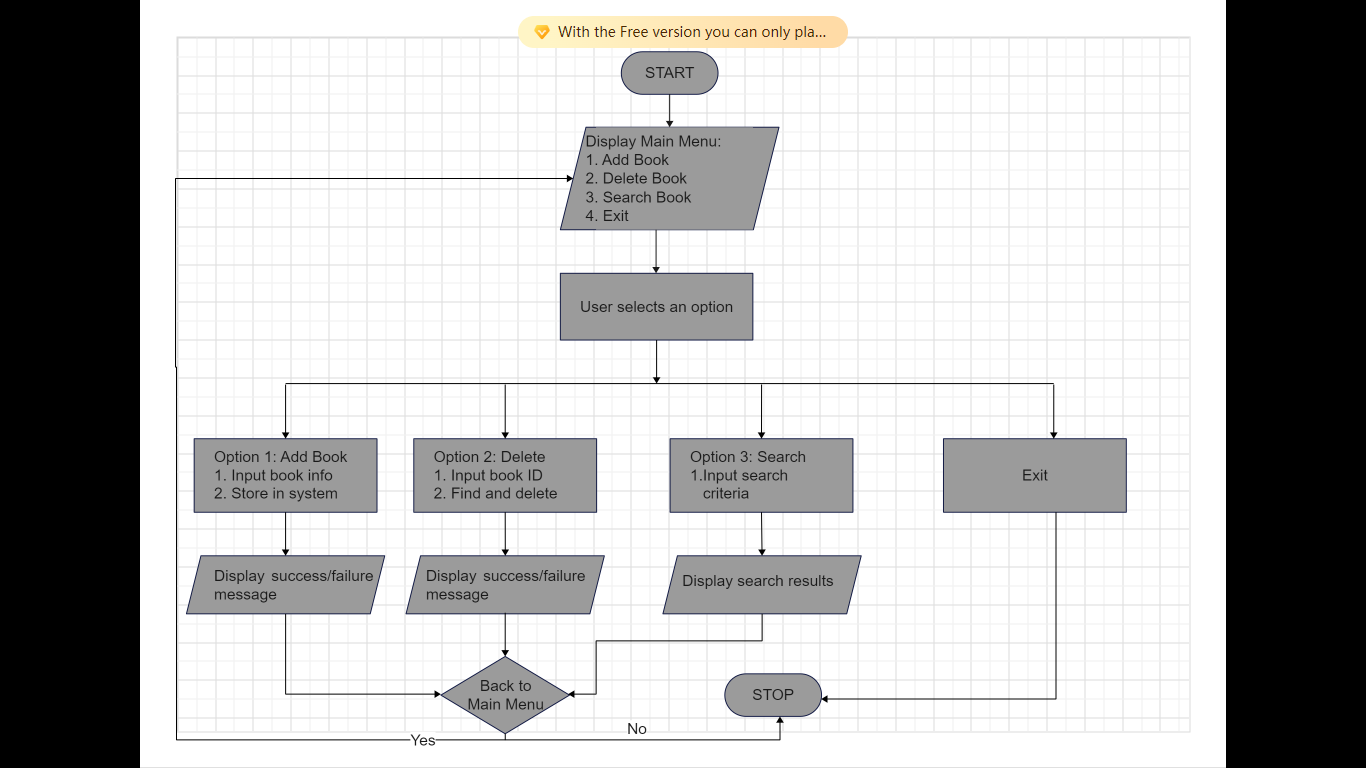
**Algorithm:**

1. Initialize the library database.

2. Provide options for adding, deleting, and searching books.

3. Perform the selected operations.

**Flowchart:**



**Implementation:**

#include <stdio.h>

#include <string.h>

// Define book structure

struct Book {

    int id;

    char title[100];

    char author[100];

    int year;

};

// Function prototypes

void addBook(struct Book books[], int \*count);

void deleteBook(struct Book books[], int \*count, int id);

void searchBook(struct Book books[], int count, int id);

int main() {

    struct Book books[100];

    int count = 0;

    int choice, id;

    while(1) {

        printf("1. Add Book\n2. Delete Book\n3. Search Book\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1: addBook(books, &count); break;

            case 2:

                printf("Enter book ID to delete: ");

                scanf("%d", &id);

                deleteBook(books, &count, id); break;

            case 3:

                printf("Enter book ID to search: ");

                scanf("%d", &id);

                searchBook(books, count, id); break;

            case 4: return 0;

            default: printf("Invalid choice!\n");

        }

    }

}

void addBook(struct Book books[], int \*count) {

    // Implementation to add a book

     if (\*count >= MAX\_BOOKS) {

        printf("Library is full, cannot add more books.\n");

        return;

    }

    // Get book details from the user

    printf("Enter Book ID: ");

    scanf("%d", &books[\*count].id);

    printf("Enter Book Title: ");

    scanf(" %[^\n]%\*c", books[\*count].title); // The format %[^\n]%\*c allows reading a line of text including spaces

    printf("Enter Author: ");

    scanf(" %[^\n]%\*c", books[\*count].author);

    printf("Enter Year of Publication: ");

    scanf("%d", &books[\*count].year);

    // Increment the count of books

    (\*count)++;

    // Confirm the book has been added

    printf("Book added successfully!\n");

}

void deleteBook(struct Book books[], int \*count, int id) {

    // Implementation to delete a book

      int i, found = 0;

    // Search for the book by ID

    for (i = 0; i < \*count; i++) {

        if (books[i].id == id) {

            found = 1;

            break;

        }

    }

    // If the book is found, delete it

    if (found) {

        // Shift the remaining books up in the array

        for (int j = i; j < \*count - 1; j++) {

            books[j] = books[j + 1];

        }

        // Decrement the book count

        (\*count)--;

        printf("Book with ID %d deleted successfully.\n", id);

    } else {

        // If the book was not found

        printf("Book with ID %d not found.\n", id);

    }

}

void searchBook(struct Book books[], int count, int id) {

    // Implementation to search for a book

    int found = 0;

    // Search for the book by ID

    for (int i = 0; i < count; i++) {

        if (books[i].id == id) {

            found = 1;

            // Display the book's details

            printf("Book found:\n");

            printf("ID: %d\n", books[i].id);

            printf("Title: %s\n", books[i].title);

            printf("Author: %s\n", books[i].author);

            printf("Year: %d\n", books[i].year);

            break;

        }

    }

    // If the book was not found

    if (!found) {

        printf("Book with ID %d not found.\n", id);

    }

}

**Screenshots**

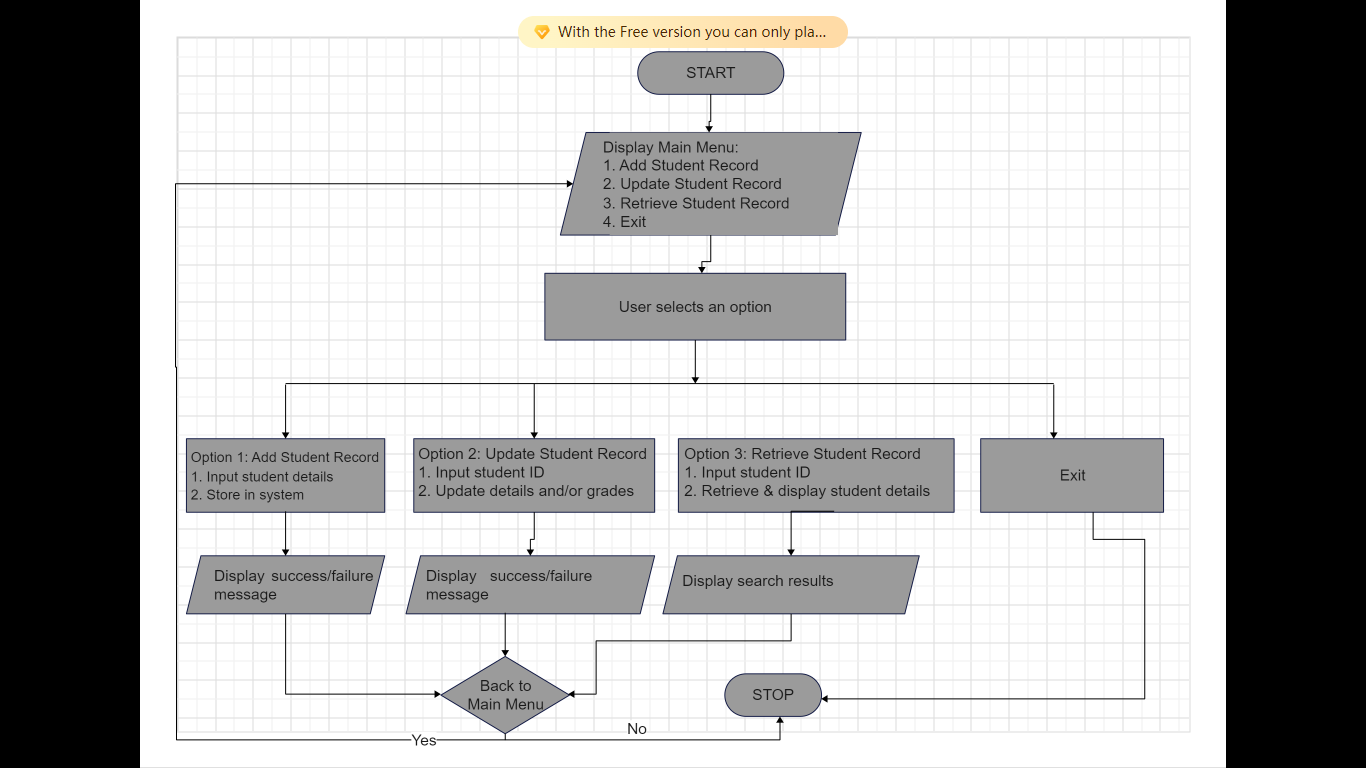
**CASE- 2: Student Database Management System**

**Problem Statement:** Create a system to manage student information, including enrollment and grades.

**Algorithm:**

1. Initialize student database.
2. Provide options to add, update, and retrieve student records.
3. Perform the selected operations.

**Flowchart:**



**Implementation:**

#include <stdio.h>

#include <string.h>

// Define student structure

struct Student {

    int id;

    char name[50];

    float grade;

};

// Function prototypes

void addStudent(struct Student students[], int \*count);

void updateStudent(struct Student students[], int count, int id);

void retrieveStudent(struct Student students[], int count, int id);

int main() {

    struct Student students[100];

    int count = 0;

    int choice, id;

    while(1) {

        printf("1. Add Student\n2. Update Student\n3. Retrieve Student\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1: addStudent(students, &count); break;

            case 2:

                printf("Enter student ID to update: ");

                scanf("%d", &id);

                updateStudent(students, count, id); break;

            case 3:

                printf("Enter student ID to retrieve: ");

                scanf("%d", &id);

                retrieveStudent(students, count, id); break;

            case 4: return 0;

            default: printf("Invalid choice!\n");

        }

    }

}

void addStudent(struct Student students[], int \*count) {

    printf("Enter student ID: ");

    scanf("%d", &students[\*count].id);

    printf("Enter student name: ");

    scanf("%s", students[\*count].name);

    printf("Enter student grade: ");

    scanf("%f", &students[\*count].grade);

    (\*count)++;

    printf("Student added successfully!\n");

}

void updateStudent(struct Student students[], int count, int id) {

    int found = 0;

    for (int i = 0; i < count; i++) {

        if (students[i].id == id) {

            found = 1;

            printf("Enter new student name: ");

            scanf("%s", students[i].name);

            printf("Enter new student grade: ");

            scanf("%f", &students[i].grade);

            printf("Student record updated successfully!\n");

            break;

        }

    }

    if (!found) {

        printf("Student with ID %d not found.\n", id);

    }

}

void retrieveStudent(struct Student students[], int count, int id) {

    int found = 0;

    for (int i = 0; i < count; i++) {

        if (students[i].id == id) {

            found = 1;

            printf("Student ID: %d\n", students[i].id);

            printf("Student Name: %s\n", students[i].name);

            printf("Student Grade: %.2f\n", students[i].grade);

            break;

        }

    }

    if (!found) {

        printf("Student with ID %d not found.\n", id);

    }

}

### ****Explanation:****

1. **addStudent Function**:
   * This function adds a new student to the array.
   * It prompts the user to enter the student’s ID, name, and grade.
   * The student is then added to the array, and the student count is incremented.
2. **updateStudent Function**:
   * This function updates the information of an existing student based on their ID.
   * It searches the array for the student with the given ID.
   * If found, it allows the user to update the student's name and grade.
   * If the student is not found, it informs the user.
3. **retrieveStudent Function**:
   * This function retrieves and displays the details of a student based on their ID.
   * It searches the array for the student with the given ID.
   * If found, it prints the student's ID, name, and grade.
   * If the student is not found, it informs the user.
4. **Main Loop**:
   * The main loop provides a menu-driven interface that repeatedly prompts the user to select an option (Add, Update, Retrieve, or Exit).
   * Depending on the user’s choice, the corresponding function is called.

**Screenshots**

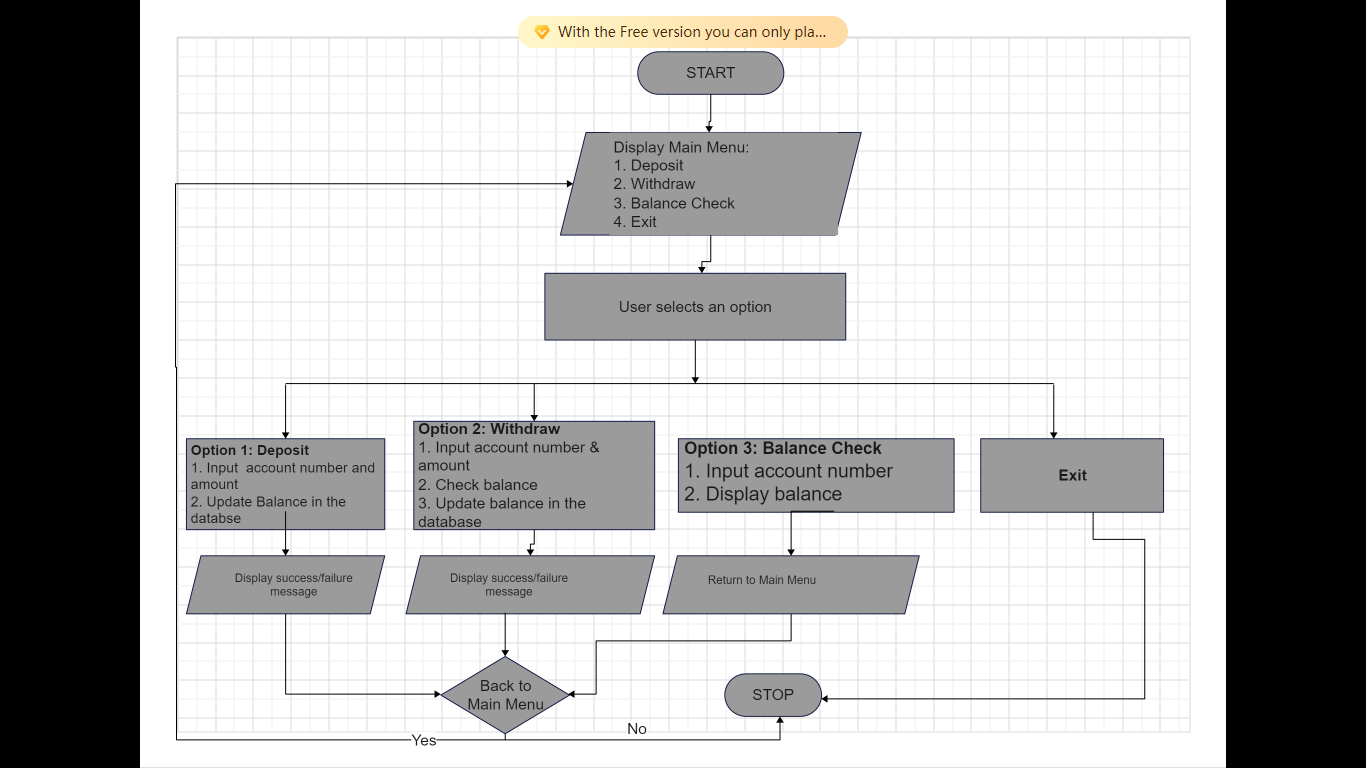
**CASE- 3: Banking System**

**Problem Statement:** Develop a system to manage bank accounts and transactions, including deposits, withdrawals, and balance checks.

**Algorithm:**

1. Initialize bank accounts database.
2. Provide options for deposit, withdrawal, and balance check.
3. Perform selected operations.

**Flowchart:**



**Implementation:**

#include <stdio.h>

// Define account structure

struct Account {

    int accountNumber;

    float balance;

};

// Function prototypes

void deposit(struct Account \*account, float amount);

void withdraw(struct Account \*account, float amount);

void checkBalance(struct Account account);

int main() {

    struct Account account = {12345, 1000.0};

    int choice;

    float amount;

    while(1) {

        printf("1. Deposit\n2. Withdraw\n3. Check Balance\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1:

                printf("Enter amount to deposit: ");

                scanf("%f", &amount);

                deposit(&account, amount); break;

            case 2:

                printf("Enter amount to withdraw: ");

                scanf("%f", &amount);

                withdraw(&account, amount); break;

            case 3: checkBalance(account); break;

            case 4: return 0;

            default: printf("Invalid choice!\n");

        }

    }

}

void deposit(struct Account \*account, float amount) {

    if (amount > 0) {

        account->balance += amount;

        printf("Successfully deposited %.2f. New balance is %.2f\n", amount, account->balance);

    } else {

        printf("Deposit amount must be positive!\n");

    }

}

void withdraw(struct Account \*account, float amount) {

    if (amount > 0 && amount <= account->balance) {

        account->balance -= amount;

        printf("Successfully withdrew %.2f. New balance is %.2f\n", amount, account->balance);

    } else if (amount > account->balance) {

        printf("Insufficient balance!\n");

    } else {

        printf("Withdrawal amount must be positive!\n");

    }

}

void checkBalance(struct Account account) {

    printf("The balance for account number %d is %.2f\n", account.accountNumber, account.balance);

}

### ****Explanation:****

1. **deposit Function**:
   * This function takes a pointer to an Account structure and the deposit amount.
   * It checks if the deposit amount is positive.
   * If valid, the amount is added to the account's balance.
   * It then displays the updated balance.
2. **withdraw Function**:
   * This function takes a pointer to an Account structure and the withdrawal amount.
   * It checks if the withdrawal amount is positive and less than or equal to the current balance.
   * If the conditions are met, the amount is subtracted from the account's balance.
   * It then displays the updated balance.
   * If the withdrawal amount is more than the balance, it displays an "Insufficient balance" message.
3. **checkBalance Function**:
   * This function takes an Account structure as an argument.
   * It displays the current balance of the account.
4. **Main Function**:
   * The main function provides a menu-driven interface for the user to choose between deposit, withdrawal, balance check, or exit.
   * It uses a loop to continuously offer these options until the user decides to exit.

This implementation allows basic operations on a bank account, such as depositing money, withdrawing money, and checking the balance, with error handling for invalid inputs.

**Screenshots**

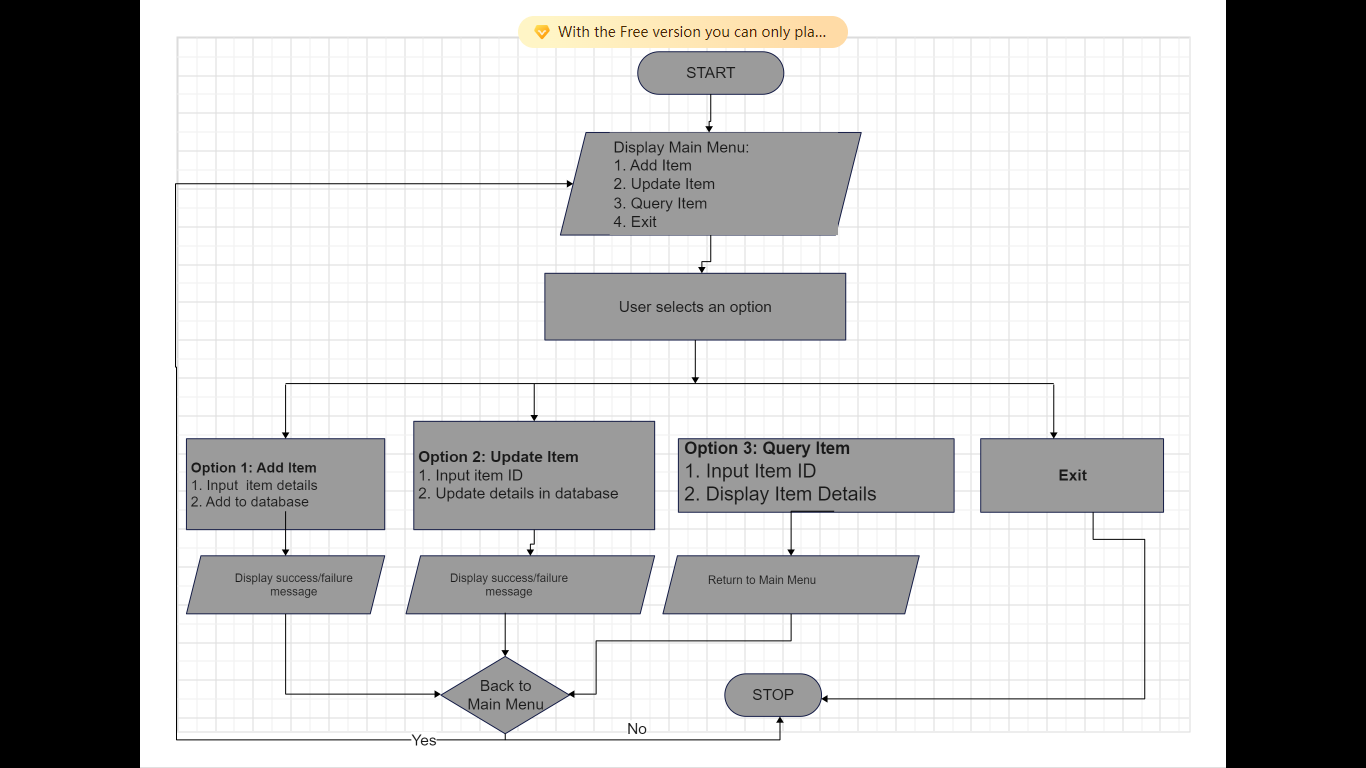
**CASE- 4: Inventory Management System**

**Problem Statement:** Build a system to manage inventory for a store, tracking items, quantities, and prices..

**Algorithm:**

1. Initialize inventory database.
2. Provide options to add, update, and query items.
3. Perform selected operations.

**Flowchart:**



**Implementation:**

#include <stdio.h>

#include <string.h>

// Define item structure

struct Item {

    int id;

    char name[50];

    int quantity;

    float price;

};

// Function prototypes

void addItem(struct Item items[], int \*count);

void updateItem(struct Item items[], int count, int id);

void queryItem(struct Item items[], int count, int id);

int main() {

    struct Item items[100];

    int count = 0;

    int choice, id;

    while(1) {

        printf("1. Add Item\n2. Update Item\n3. Query Item\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1: addItem(items, &count); break;

            case 2:

                printf("Enter item ID to update: ");

                scanf("%d", &id);

                updateItem(items, count, id); break;

            case 3:

                printf("Enter item ID to query: ");

                scanf("%d", &id);

                queryItem(items, count, id); break;

            case 4: return 0;

            default: printf("Invalid choice!\n");

        }

    }

}

void addItem(struct Item items[], int \*count) {

    struct Item newItem;

    newItem.id = \*count + 1;  // Automatically assign an ID

    printf("Enter item name: ");

    scanf("%s", newItem.name);

    printf("Enter item quantity: ");

    scanf("%d", &newItem.quantity);

    printf("Enter item price: ");

    scanf("%f", &newItem.price);

    items[\*count] = newItem;

    (\*count)++;

    printf("Item added successfully! ID: %d\n", newItem.id);

}

void updateItem(struct Item items[], int count, int id) {

    int found = 0;

    for (int i = 0; i < count; i++) {

        if (items[i].id == id) {

            found = 1;

            printf("Enter new item name: ");

            scanf("%s", items[i].name);

            printf("Enter new item quantity: ");

            scanf("%d", &items[i].quantity);

            printf("Enter new item price: ");

            scanf("%f", &items[i].price);

            printf("Item updated successfully!\n");

            break;

        }

    }

    if (!found) {

        printf("Item with ID %d not found!\n", id);

    }

}

void queryItem(struct Item items[], int count, int id) {

    int found = 0;

    for (int i = 0; i < count; i++) {

        if (items[i].id == id) {

            found = 1;

            printf("Item ID: %d\n", items[i].id);

            printf("Item Name: %s\n", items[i].name);

            printf("Item Quantity: %d\n", items[i].quantity);

            printf("Item Price: %.2f\n", items[i].price);

            break;

        }

    }

    if (!found) {

        printf("Item with ID %d not found!\n", id);

    }

}

**Explanation:**

1. **addItem Function**:
   * A new item is created with a unique ID automatically assigned based on the current count.
   * The user is prompted to enter the item name, quantity, and price.
   * The item is added to the array, and the count is incremented.
   * A success message is displayed with the item's ID.
2. **updateItem Function**:
   * The user is prompted to enter the ID of the item to update.
   * The system searches for the item by ID.
   * If found, the user can update the name, quantity, and price of the item.
   * If the item is found, a success message is displayed. If not, an error message is shown.
3. **queryItem Function**:
   * The user is prompted to enter the ID of the item to query.
   * The system searches for the item by ID.
   * If found, the item’s details (ID, name, quantity, price) are displayed.
   * If not, an error message is shown.

### ****Main Function****:

The main function provides a loop that allows users to continually add, update, or query items until they choose to exit. This simple system handles basic inventory management tasks effectively.

**Screenshots**:

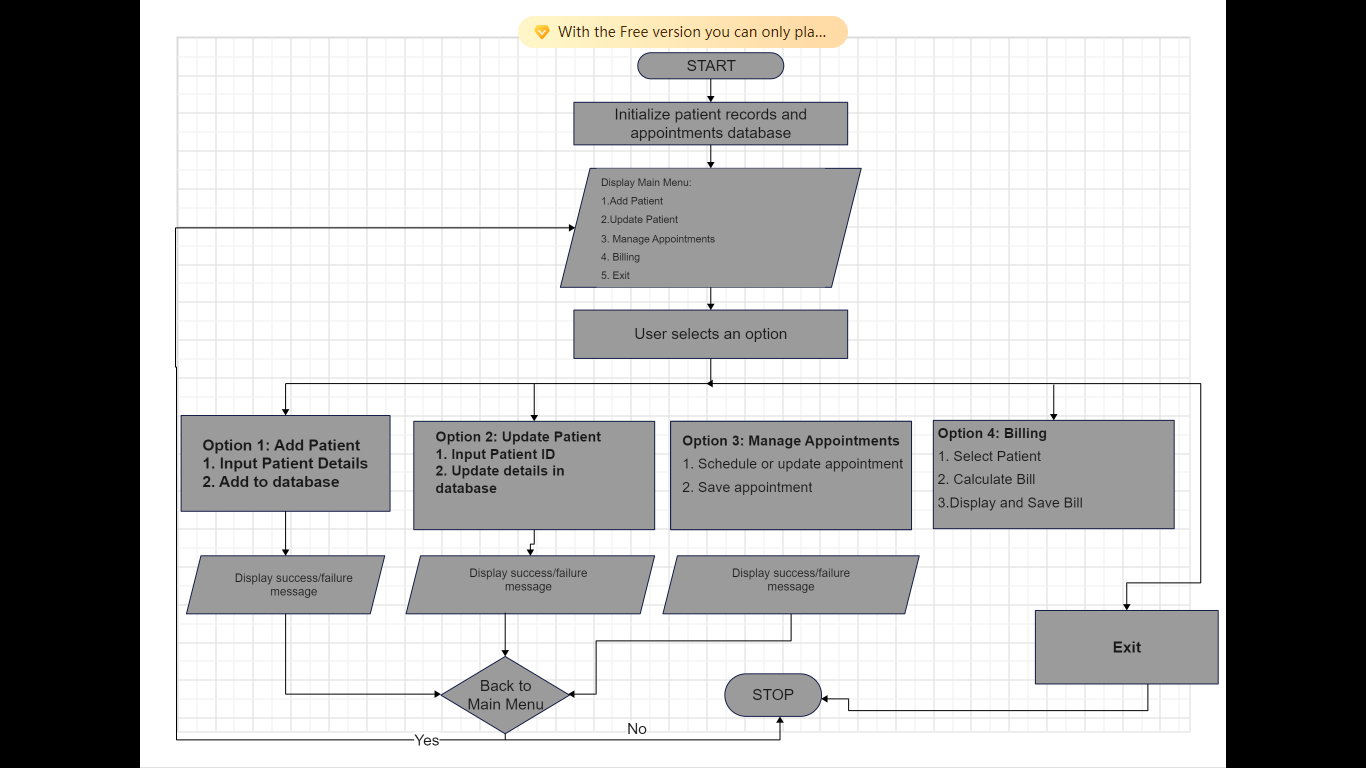
**CASE- 5: Hospital Management System**

**Problem Statement:** Create a system for managing patient records, appointments, and billing.

**Algorithm:**

1. Initialize patient records database.
2. Provide options to add, update, and manage appointments.
3. Perform selected operations.

**Flowchart:**



**Implementation:**

#include <stdio.h>

#include <string.h>

// Define patient structure

struct Patient {

    int id;

    char name[50];

    char contact[15];

};

// Define appointment structure

struct Appointment {

    int patientId;

    char date[15];

    char time[10];

};

// Function prototypes

void addPatient(struct Patient patients[], int \*patientCount);

void updatePatient(struct Patient patients[], int patientCount);

void manageAppointments(struct Appointment appointments[], int \*appointmentCount, int patientCount);

void billing(struct Patient patients[], int patientCount);

int main() {

    struct Patient patients[100];

    struct Appointment appointments[100];

    int patientCount = 0;

    int appointmentCount = 0;

    int choice;

    while(1) {

        printf("\n--- Main Menu ---\n");

        printf("1. Add Patient\n");

        printf("2. Update Patient\n");

        printf("3. Manage Appointments\n");

        printf("4. Billing\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1:

                addPatient(patients, &patientCount);

                break;

            case 2:

                updatePatient(patients, patientCount);

                break;

            case 3:

                manageAppointments(appointments, &appointmentCount, patientCount);

                break;

            case 4:

                billing(patients, patientCount);

                break;

            case 5:

                printf("Exiting system...\n");

                return 0;

            default:

                printf("Invalid choice! Please try again.\n");

        }

    }

}

void addPatient(struct Patient patients[], int \*patientCount) {

    struct Patient newPatient;

    newPatient.id = \*patientCount + 1;  // Automatically assign an ID

    printf("Enter patient name: ");

    scanf("%s", newPatient.name);

    printf("Enter patient contact: ");

    scanf("%s", newPatient.contact);

    patients[\*patientCount] = newPatient;

    (\*patientCount)++;

    printf("Patient added successfully! ID: %d\n", newPatient.id);

}

void updatePatient(struct Patient patients[], int patientCount) {

    int id;

    int found = 0;

    printf("Enter patient ID to update: ");

    scanf("%d", &id);

    for (int i = 0; i < patientCount; i++) {

        if (patients[i].id == id) {

            found = 1;

            printf("Enter new patient name: ");

            scanf("%s", patients[i].name);

            printf("Enter new patient contact: ");

            scanf("%s", patients[i].contact);

            printf("Patient updated successfully!\n");

            break;

        }

    }

    if (!found) {

        printf("Patient with ID %d not found!\n", id);

    }

}

void manageAppointments(struct Appointment appointments[], int \*appointmentCount, int patientCount) {

    if (patientCount == 0) {

        printf("No patients available to schedule an appointment.\n");

        return;

    }

    struct Appointment newAppointment;

    printf("Enter patient ID for the appointment: ");

    scanf("%d", &newAppointment.patientId);

    printf("Enter appointment date (dd-mm-yyyy): ");

    scanf("%s", newAppointment.date);

    printf("Enter appointment time (hh:mm): ");

    scanf("%s", newAppointment.time);

    appointments[\*appointmentCount] = newAppointment;

    (\*appointmentCount)++;

    printf("Appointment scheduled successfully!\n");

}

void billing(struct Patient patients[], int patientCount) {

    int id;

    int found = 0;

    printf("Enter patient ID for billing: ");

    scanf("%d", &id);

    for (int i = 0; i < patientCount; i++) {

        if (patients[i].id == id) {

            found = 1;

            printf("Patient ID: %d\n", patients[i].id);

            printf("Patient Name: %s\n", patients[i].name);

            printf("Enter total bill amount: ");

            float amount;

            scanf("%f", &amount);

            printf("Billing successful! Amount: %.2f\n", amount);

            break;

        }

    }

    if (!found) {

        printf("Patient with ID %d not found!\n");

    }

}

**Explanation :**

 **Data Structures**:

* **Patient**: Stores the patient's ID, name, and contact information.
* **Appointment**: Stores the patient's ID, the date, and the time of the appointment.

 **Functionality**:

* **addPatient**: Adds a new patient to the system. The patient is automatically assigned an ID based on the current count of patients.
* **updatePatient**: Allows updating a patient's details by searching with the patient's ID.
* **manageAppointments**: Manages the appointments by scheduling new appointments with the patient’s ID.
* **billing**: Handles the billing process by searching for a patient using their ID and allowing the user to enter the total bill amount.

 **User Interface**:

* The main function continuously displays a menu that allows the user to choose different operations (adding, updating patients, managing appointments, billing).
* After selecting an operation, the respective function is called, and the process is carried out.

 **Termination**:

* The system exits when the user selects the option to exit.

**Screenshots** :

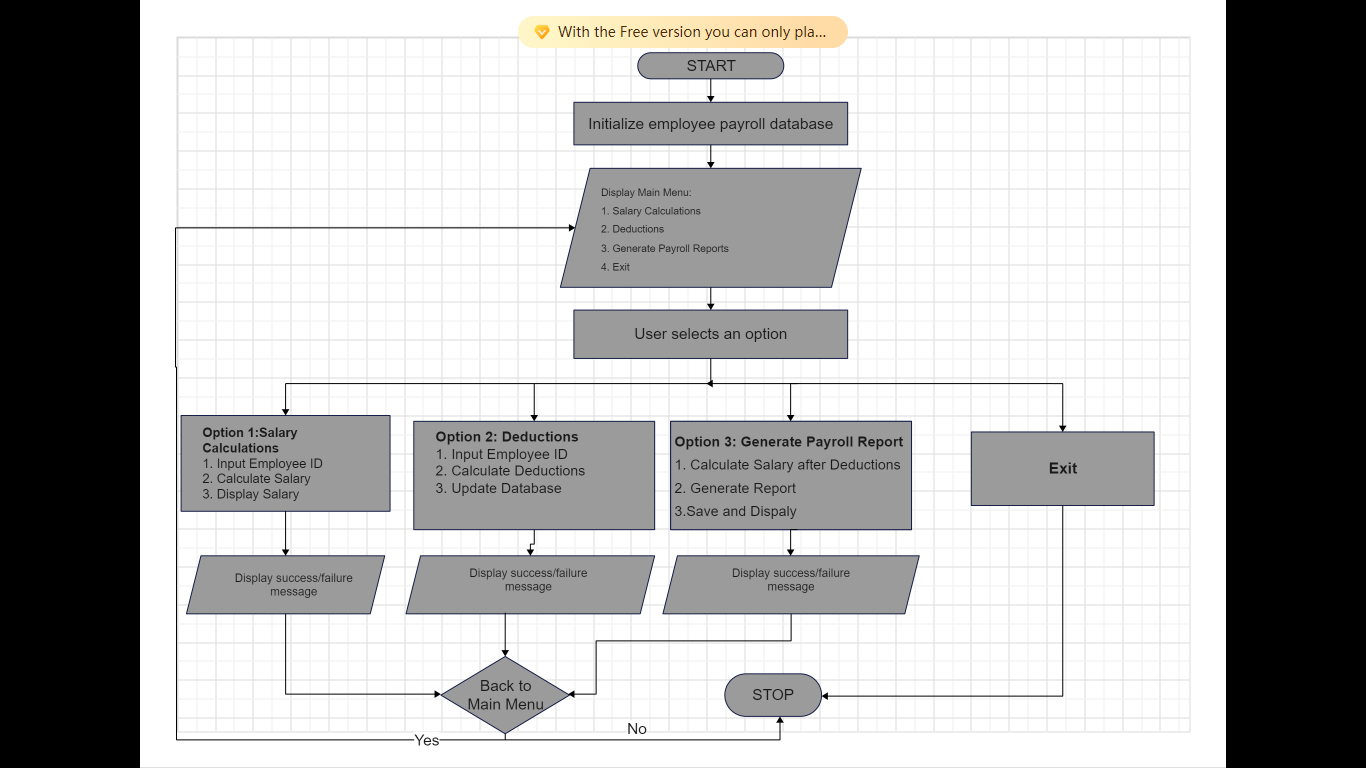
**CASE- 6: Employee Payroll System**

**Problem Statement:** Develop a system to manage employee payroll, including salary calculations and deductions.

**Algorithm:**

1. Initialize employee payroll database.
2. Provide options for salary calculations and deductions.
3. Generate payroll reports.

**Flowchart:**



**Implementation:**

#include <stdio.h>

// Define employee structure

struct Employee {

    int id;

    char name[50];

    float basicSalary;

    float deductions;

    float netSalary;

};

// Function prototypes

void calculateSalary(struct Employee \*employee);

void applyDeductions(struct Employee \*employee);

void generatePayrollReport(struct Employee employees[], int count);

int main() {

    struct Employee employees[100];

    int count = 0;

    int choice, id;

    while(1) {

        printf("\n--- Main Menu ---\n");

        printf("1. Salary Calculations\n");

        printf("2. Deductions\n");

        printf("3. Generate Payroll Report\n");

        printf("4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch(choice) {

            case 1:

                printf("Enter employee ID: ");

                scanf("%d", &id);

                calculateSalary(&employees[id - 1]);

                break;

            case 2:

                printf("Enter employee ID: ");

                scanf("%d", &id);

                applyDeductions(&employees[id - 1]);

                break;

            case 3:

                generatePayrollReport(employees, count);

                break;

            case 4:

                printf("Exiting system...\n");

                return 0;

            default:

                printf("Invalid choice! Please try again.\n");

        }

    }

}

void calculateSalary(struct Employee \*employee) {

    if(employee->id == 0) {

        printf("Employee not found. Please add the employee first.\n");

        return;

    }

    printf("Enter basic salary for %s: ", employee->name);

    scanf("%f", &employee->basicSalary);

    employee->netSalary = employee->basicSalary - employee->deductions;

    printf("Salary calculated for %s: %.2f\n", employee->name, employee->netSalary);

}

void applyDeductions(struct Employee \*employee) {

    if(employee->id == 0) {

        printf("Employee not found. Please add the employee first.\n");

        return;

    }

    printf("Enter deductions for %s: ", employee->name);

    scanf("%f", &employee->deductions);

    employee->netSalary = employee->basicSalary - employee->deductions;

    printf("Deductions applied for %s: %.2f\n", employee->name, employee->deductions);

}

void generatePayrollReport(struct Employee employees[], int count) {

    printf("\n--- Payroll Report ---\n");

    for(int i = 0; i < count; i++) {

        if(employees[i].id != 0) {

            printf("Employee ID: %d\n", employees[i].id);

            printf("Name: %s\n", employees[i].name);

            printf("Basic Salary: %.2f\n", employees[i].basicSalary);

            printf("Deductions: %.2f\n", employees[i].deductions);

            printf("Net Salary: %.2f\n\n", employees[i].netSalary);

        }

    }

}

**Explanations :**

 **Data Structures**:

* **Employee**: Stores employee ID, name, basic salary, deductions, and net salary.

 **Functionality**:

* **calculateSalary**: Calculates the employee's net salary based on their basic salary and any deductions.
* **applyDeductions**: Applies deductions to the employee’s salary, such as taxes or insurance, and updates the net salary.
* **generatePayrollReport**: Generates a payroll report for all employees, displaying their salary details.

 **User Interface**:

* The main function displays a menu with options to calculate salary, apply deductions, and generate payroll reports.
* After the user selects an option, the corresponding function is executed.

 **Employee Addition**:

* This code assumes that employees are already added to the system, indexed by their ID.
* If the employee does not exist in the system, a message prompts the user to add them first.

 **Termination**:

* The system continues to prompt the user for operations until they choose to exit.

**Screenshots** :

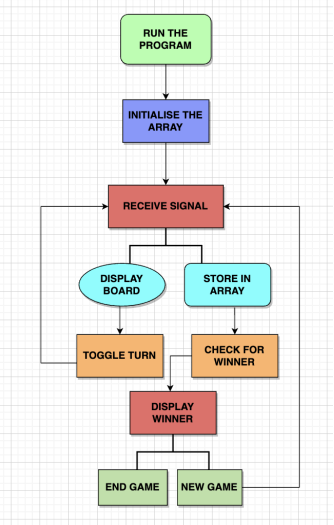
**CASE- 7: Tic-Tac-Toe Game**

**Problem Statement:** Implement a console-based Tic-Tac-Toe game.

**Algorithm:**

1. **Initialize the Array:**
   * Create a 2D array to represent the Tic-Tac-Toe board, initially filled with empty spaces.
2. **Receive the Signal:**
   * Wait for the player to input their move, which consists of a row and column selection.
3. **Display the Board:**
   * Print the current state of the board to the console, showing each cell's content.
4. **Toggle Turn:**
   * Switch between Player 1 and Player 2 after each move.
5. **Store in Array:**
   * Place the player's mark (X or O) in the selected position on the board.
6. **Check for Winner:**
   * After each move, check all rows, columns, and diagonals to see if there is a winning combination.
7. **Display Winner:**
   * If a player wins, display the winner’s name and end the game.
8. **End Game:**
   * If the board is full or a player wins, offer the option to end the game.
9. **New Game:**
   * If the players choose to play again, reinitialize the board and start a new game.

**Flowchart:**



**Implementation:**

#include <stdio.h>

#define SIZE 3

// Function prototypes

void initializeBoard(char board[SIZE][SIZE]);

void displayBoard(char board[SIZE][SIZE]);

int checkWin(char board[SIZE][SIZE]);

int isBoardFull(char board[SIZE][SIZE]);

void makeMove(char board[SIZE][SIZE], int player);

void togglePlayer(int \*player);

void newGame(char board[SIZE][SIZE], int \*player);

int main() {

    char board[SIZE][SIZE];

    int currentPlayer = 1;

    int winner = 0;

    char playAgain;

    do {

        newGame(board, &currentPlayer);

        while(1) {

            displayBoard(board);

            makeMove(board, currentPlayer);

            // Check if the current player wins

            winner = checkWin(board);

            if (winner) {

                displayBoard(board);

                printf("Player %d wins!\n", currentPlayer);

                break;

            }

            // Check if the board is full (draw)

            if (isBoardFull(board)) {

                displayBoard(board);

                printf("It's a draw!\n");

                break;

            }

            // Toggle player

            togglePlayer(&currentPlayer);

        }

        // Ask if the players want to play again

        printf("Do you want to play again? (y/n): ");

        scanf(" %c", &playAgain);

    } while(playAgain == 'y' || playAgain == 'Y');

    return 0;

}

// Function to initialize the game board

void initializeBoard(char board[SIZE][SIZE]) {

    for (int i = 0; i < SIZE; i++) {

        for (int j = 0; j < SIZE; j++) {

            board[i][j] = ' ';

        }

    }

}

// Function to display the game board

void displayBoard(char board[SIZE][SIZE]) {

    printf("\n");

    for (int i = 0; i < SIZE; i++) {

        for (int j = 0; j < SIZE; j++) {

            printf(" %c ", board[i][j]);

            if (j < SIZE - 1) printf("|");

        }

        printf("\n");

        if (i < SIZE - 1) printf("---+---+---\n");

    }

    printf("\n");

}

// Function to check if a player has won

int checkWin(char board[SIZE][SIZE]) {

    // Check rows and columns

    for (int i = 0; i < SIZE; i++) {

        if ((board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] != ' ') ||

            (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] != ' ')) {

            return 1;

        }

    }

    // Check diagonals

    if ((board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] != ' ') ||

        (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] != ' ')) {

        return 1;

    }

    return 0;

}

// Function to check if the board is full

int isBoardFull(char board[SIZE][SIZE]) {

    for (int i = 0; i < SIZE; i++) {

        for (int j = 0; j < SIZE; j++) {

            if (board[i][j] == ' ') return 0;

        }

    }

    return 1;

}

// Function to make a move for the current player

void makeMove(char board[SIZE][SIZE], int player) {

    int row, col;

    char mark = (player == 1) ? 'X' : 'O';

    while (1) {

        printf("Player %d, enter row (1-3) and column (1-3): ", player);

        scanf("%d %d", &row, &col);

        row--; col--;  // Convert to 0-based index

        if (row >= 0 && row < SIZE && col >= 0 && col < SIZE && board[row][col] == ' ') {

            board[row][col] = mark;

            break;

        } else {

            printf("Invalid move! Try again.\n");

        }

    }

}

// Function to toggle between players

void togglePlayer(int \*player) {

    \*player = (\*player == 1) ? 2 : 1;

}

// Function to start a new game

void newGame(char board[SIZE][SIZE], int \*player) {

    initializeBoard(board);

    \*player = 1;

}

**Explanation**

1. **Initialization**:
   * The initializeBoard function sets up an empty game board.
   * The newGame function resets the board and sets Player 1 as the starting player.
2. **Main Game Loop**:
   * The game enters a loop where the board is displayed using displayBoard.
   * Players alternate turns to make a move using the makeMove function.
   * After each move, checkWin checks for a win condition, and isBoardFull checks for a draw.
   * If a player wins or the game ends in a draw, the loop breaks, and the winner or draw is announced.
3. **Toggling Between Players**:
   * The togglePlayer function switches between Player 1 and Player 2 after each valid move.
4. **Play Again Option**:
   * After each game, the program prompts the users if they want to play again.
   * If they choose "y", the game restarts with a new game board.

### ****Running the Program****:

* When run, the program will continuously execute games until the players decide to stop by inputting 'n' when asked if they want to play again.

**Screenshots** :

**CASE- 8: Sorting and Searching Algorithms**

**Problem Statement:** Implement and compare various sorting and searching algorithms (e.g., Bubble Sort, Quick Sort, Binary Search).

**Algorithm:**

1. Implement sorting algorithms: Bubble Sort, Quick Sort.
2. Implement searching algorithm: Binary Search.
3. Compare performance of each algorithm.

**Flowchart:**

**Implementation:**

#include <stdio.h>

void bubbleSort(int arr[], int n);

void quickSort(int arr[], int low, int high);

int binarySearch(int arr[], int size, int target);

int main() {

    // Example usage of sorting and searching algorithms

    return 0;

}

void bubbleSort(int arr[], int n) {

    // Implementation of Bubble Sort

}

void quickSort(int arr[], int low, int high) {

    // Implementation of Quick Sort

}

int binarySearch(int arr[], int size, int target) {

    // Implementation of Binary Search

    return -1;

}

**Screenshots** :

### ****Conclusion****

The internship provided an opportunity to develop practical skills in C programming by working on diverse case studies. Each project demonstrated the application of various programming concepts, including data structures, algorithms, and user interface design. The projects ranged from system management and games to sorting algorithms, showcasing the versatility of C programming. Through these case studies, significant insights were gained into problem-solving and software development practices, reinforcing theoretical knowledge with practical experience.