*A report on*

# Event Management System using C

*submitted in partial fulfillment of the requirements for completion of*

**Data Structures and Algorithms - 2 Lab**

*of*

## SY COMP

*in*

## Computer Engineering

*by*


**Sumit Vitthal Desai, Piyush Deshpande and Bhavesh Gadling**

612415049, 612415052 and 612415061


*Under the guidance of*


**Prof. Tina Francis**

*Professor*

*Department of Computer Engineering*

**Department of Computer Engineering,**

**COEP Technological University (COEP Tech)**

(A Unitary Public University of Govt. of Maharashtra)

Shivajinagar, Pune-411005, Maharashtra, INDIA


**December 2025**

# Contents

# 1 Introduction

Event management—whether in universities, corporations, or community organizations—requires careful coordination of schedules, venues, and participants. The system described in this report is a compact, console-based Event Management System (EMS) implemented in C. It is built to illustrate how classical data structures and algorithms can be applied to real-world problems: we chose each data structure where it naturally solves a core problem rather than forcing a tool to fit.

This EMS integrates three principal modules:

- **Login and Registration Module:** Handles user authentication, maintains role information (organizer/attendee), and exposes a simple API for other modules.
- **Event Management Module:** Creates, updates, and deletes events; enforces scheduling rules and venue constraints.
- **Attendee Management Module:** Manages registrations, attendance marking, and attendee-level reporting.

Each module communicates through a shared persistence layer (CSV files) and well-defined function interfaces. The aim is to keep the core clean and modular so the system can later be extended with a graphical or web front-end without changing the essential algorithms.

## 1.1 Definitions

To avoid ambiguity, the central terms used in this report are defined succinctly:

- **Event:** A scheduled occurrence identified by an eventID and described by attributes such as title, date, start and end times, venue, registration deadline, and maximum capacity.
- **Venue:** A named location (physical or virtual) with an identifier, capacity, and availability schedule.
- **Organizer:** A user role that can create or manage events and view attendee lists.
- **Attendee:** A user who may register for events and whose attendance can be recorded.
- **Registration:** A record linking an attendee to an event; it can be created, cancelled, or updated.

These definitions inform the data model and the function interfaces we implement.

## 1.2 Applications

The EMS can be deployed in multiple contexts. We outline four typical application domains and highlight relevant capabilities.

### 1.2.1 University Events

Universities manage seminars, practical sessions, and cultural events. The EMS assists by:

- Preventing classroom and auditorium double-bookings.
- Tracking attendance for credit or participation certificates.
- Producing departmental reports and historical logs.

### 1.2.2 Corporate Events

For organizations, the EMS supports:

- Efficient meeting-room allocation and conflict avoidance.
- Attendance tracking for compliance and payroll (if needed).
- Analytics for resource usage and event popularity.

### 1.2.3 Social Gatherings

Community and social event organizers benefit from:

- Simple registration flows for members.
- Exportable attendee lists for on-site check-in.
- Lightweight reporting for future planning.

### 1.2.4 Technical Conferences

Conferences require multi-track scheduling and scalability:

- Parallel sessions with capacity and speaker coordination.
- Conflict-free scheduling using interval trees for overlap checks.
- Attendance certificates and session analytics.

# 2 Data Structures used

Choosing the right data structure for each subsystem is fundamental to building an efficient EMS. The subsequent subsections explain our chosen structures, their justification, and their operational semantics.

## 2.1 AVL Tree

### 2.1.1 The AVL tree Data Structure

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the absolute difference between heights of left and right subtrees for any node cannot be more than one.

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

In an AVL Tree, the balance factors of all the nodes has to be strictly -1, 0 or 1.

### 2.1.2 Why AVL Trees?

Using AVL trees can significantly improve the efficiency of operations like searching, inserting, and deleting data.

1. **Efficient Searching:** AVL trees are balanced, meaning the depth of the tree is kept to a minimum. This ensures that searching for a specific item in the tree is fast, even as the tree grows larger.

2. **Balanced Structure:** Unlike regular binary search trees, AVL trees automatically balance themselves after each operation. This prevents the tree from becoming lopsided or skewed, which could slow down operations.

3. **Predictable Performance:** Because AVL trees maintain their balance, their performance is consistent. Regardless of the order in which data is inserted, the tree maintains its optimal structure, leading to predictable and reliable performance.

4. **Optimized for Event Management:** AVL trees are commonly used in applications where fast searching and sorting are essential, such as event management systems. Their efficient structure makes them ideal for handling large volumes of events.

### 2.1.3 Worst case Time complexities of AVL Tree

- **Search/Lookup:** In the worst case, the search operation in an AVL tree requires traversing from the root to a leaf node along a path of height $h$, where $h$ is the height of the tree. Since AVL trees are balanced and their height is $O(\lg n)$, where $n$ is the number of nodes, the worst-case time complexity for search is $O(\lg n)$.

- **Insertion:** $O(\lg n)$

- **Deletion:** $O(\lg n)$
- **Finding maximum/minimum element:** $O(\lg n)$

## 2.2 Singly Linked List

### 2.2.1 The Singly Linked List Data Structure

A singly linked list stores nodes that hold data and a pointer to the next node. For attendees, the node encapsulates the attendee record; for small sequences of events, it can also be used for ordered traversal or temporary lists.

```
typedef struct Attendee {
    int attendeeID;
    char name[50];
    char email[50];
    unsigned long long phoneNo;
    int eventID;
    int eventsRegistered;
    char status[15]; // present, absent or only registered
    char registrationDate[30];
} Attendee;

typedef struct Node {
    Attendee data;
    struct Node* next;
} Node;
```

Listing 1: Attendee and node definitions from attendee.h

### 2.2.2 Why Singly Linked List?

The primary reasons for choosing linked lists for attendee storage are:

- **Dynamic size:** The number of attendees varies unpredictably; linked lists grow as required with no pre-allocation.
- **Fast head insertion:** Adding recent registrations at the head is an $O(1)$ operation.
- **Simple persistence mapping:** Each node translates naturally to a CSV row; loading simply reconstructs the list.
- **Memory efficiency:** Only required nodes are allocated, reducing wasted space.

### 2.2.3 Worst case Time complexities of Singly Linked List

- Insertion at head: $O(1)$
- Search: $O(n)$
- Deletion (search + remove): $O(n)$
- Traversal (reporting): $O(n)$

## 2.3 Hash Table

### 2.3.1 Hash Functions Used

Venues are stored in a hash table using the modulus method:

$$\text{index} = \text{venueID} \% 101$$

We chose 101 (a prime) to minimize clustering for typical small venue sets.

### 2.3.2 Collision Handling Strategy

Separate chaining is used; each bucket is a linked list of venues. The operations are:

- **Insertion** — compute index, prepend to chain ($O(1)$ average)
- **Lookup** — traverse short chain ($O(1 + \alpha)$ average)
- **Deletion** — remove from chain with pointer update ($O(1)$ after traversal)

This approach is simple and robust for a small number of venues.

## 2.4 Interval Tree

### 2.4.1 Interval Tree Properties

To detect schedule conflicts efficiently, we use interval trees built for a particular venue and date. Each interval node stores the start and end times (converted to integer seconds) and the maximum end time of its subtree to allow efficient pruning during overlap queries.

### 2.4.2 Use Cases in Event Scheduling

When adding a new event, the system:

1. Loads all events for the venue and date.
2. Creates an interval tree keyed by start times.

3. Queries the tree for overlap with the candidate event; if any overlap exists, booking is rejected.

This reduces checking from $O(p^2)$ naive comparisons to $O(p \log p)$ for building and substantially fewer comparisons during querying.

# 3 Time Complexity Analysis of Various Algorithms

This section provides comprehensive complexity analysis of all major operations implemented in the EMS, covering linked list operations, tree operations, sorting algorithms, file operations, and module-specific functions.

## 3.1 Linked List Operations

**Attendee Management Operations:**

- **Insertion at head:** $O(1)$ - Used when registering new attendees
- **Linear search:** $O(n)$ - Used for duplicate checks, attendee lookup
- **Deletion:** $O(n)$ - Used when unregistering attendees (search + remove)
- **Traversal:** $O(n)$ - Used for viewing all attendees, statistics calculation

**Event Linked List Operations:**

- **Add to list:** $O(1)$ - Adding event nodes at head
- **Traversal for viewing:** $O(n)$ - Displaying all events
- **Clean past events:** $O(n)$ - Single pass to identify expired events

## 3.2 Tree Operations

**AVL Tree Operations (Event Management):**

- **Insertion:** $O(\log n)$ - Inserting new event with rebalancing
- **Search by ID:** $O(\log n)$ - Searching for specific event
- **Deletion:** $O(\log n)$ - Removing event with rebalancing
- **Height calculation:** $O(1)$ - Accessing stored height
- **Rotations:** $O(1)$ - All four rotation types

**Interval Tree Operations:**

- **Tree construction:** $O(m \log m)$ - Building from $m$ intervals
- **Overlap query:** $O(\log m)$ in balanced case - Checking for scheduling conflicts
- **Insertion:** $O(\log m)$ - Adding new interval to tree

## 3.3 Sorting Algorithms

**QuickSort Implementation:**

- **Average case:** $O(n \log n)$ - Used for sorting events by date/time/ID
- **Worst case:** $O(n^2)$ - Rare with proper pivot selection
- **Partition:** $O(n)$ - For each recursive call

**Comparison Functions:**

- compareChronological(): $O(1)$ - Comparing dates and times
- compareByDate(): $O(1)$ - Date comparison only
- compareByID(): $O(1)$ - Integer comparison

**Array Conversion:**

- listToArray(): $O(n)$ - Converting linked list to array for sorting

## 3.4 File Operations

**CSV File Operations:**

- **Loading attendees:** $O(n)$ - Reading event_{id}.csv
- **Saving attendees:** $O(n)$ - Writing to CSV file
- **Fetching user data:** $O(k)$ - Scanning userAttendee.csv with $k$ users
- **Updating events attended:** $O(k)$ - Updating user record with temp file

**Persistence Operations:**

- **Atomic updates:** $O(n)$ - Using temporary files and rename()
- **Parsing with strtok():** $O(m)$ per line - Robust CSV parsing

## 3.5 Module-Specific Operations

**Login Module:**

- **User registration:** $O(k)$ - Checking existing users in CSV
- **User login:** $O(k)$ - Linear search through user database
- **Logout:** $O(1)$ - Simple status update
- **getDetails():** $O(1)$ - Returning global userStatus struct

**Attendee Module:**

- **Register attendee:** $O(k + n)$ - User fetch + attendee list operations
- **Unregister attendee:** $O(n)$ - Search and deletion
- **Mark attendance:** $O(n)$ - Linear search and update
- **View statistics:** $O(n)$ - Single traversal for counting
- **Search attendee:** $O(n)$ - Linear search by ID

**Event Module:**

- **Add event:** $O(\log m + p \log p)$ - AVL insertion + interval conflict check
- **Delete event:** $O(\log m)$ - AVL deletion
- **Modify event:** $O(\log m)$ - Search + update
- **View events:** $O(n)$ - Linked list traversal
- **Sort events:** $O(n \log n)$ - QuickSort on array
- **Generate event ID:** $O(n)$ - Linear scan for maximum ID
- **Check venue availability:** $O(p \log p)$ - Interval tree construction and query

# 4 Future Goals to be achieved

This section outlines pragmatic extensions that keep the current architecture intact while improving usability and scalability.

## 4.1 Short-Term Goals

### Improve User Interface

Upgrade the console UI with structured menus, ANSI-based coloring, and table formatting for readability. Tools and libraries: ncurses for Unix-like systems or cross-platform libraries for GUI front-ends.

### Refactor Persistence Layer

Abstract file operations behind a persistence module to allow easy substitution of CSV, serialized binary files, or database backends.

## 4.2 Long term goals

### Database Integration

Adopt SQLite for local, ACID-compliant storage or PostgreSQL/MySQL for enterprise deployments. Benefits: concurrency, indexes, transactions, and complex queries.

### RESTful API and Web Front-End

Expose the EMS functionality via an HTTP API and build a web interface (React/Vue) or a mobile app. This improves accessibility and allows multiple concurrent users.

### Advanced Analytics and Notifications

Add modules for attendance trends, event recommendations, and automated notifications (email/SMS) using simple integrations (SMTP, Twilio).

## 4.3 Limitations of the Current System

Although the EMS functions reliably for small to medium-scale usage, several limitations remain due to the simplicity of the underlying storage and interface:

- **File-based Storage:** The use of CSV files limits scalability. As the number of users or events grows, file operations become slower and more prone to data inconsistency due to partial writes or unexpected termination.

- **Lack of Concurrency Support:** Two users cannot safely modify the system at the same time. Without locking mechanisms, simultaneous updates may lead to overwritten data.
- **Basic User Interface:** The console interface, while functional, does not provide the ease and accessibility of graphical or web-based platforms.
- **Limited Security Features:** Passwords are stored in plain text and role permissions are basic. The system is not designed for high-security environments.
- **No Real-Time Notifications:** Users must manually check event information; the system does not push reminders or updates.

# 5 Appendix: Data Structures and Implementation Details

## 5.1 Complete Data Structure Definitions

```c
#ifndef EVENTS_H
#define EVENTS_H

#include <stdio.h>

typedef struct Time {
    short unsigned int hour;
    short unsigned int minute;
    short unsigned int second;
} Time;

typedef struct date {
    short int date;
    short int month;
    short int year;
} date;

typedef struct event {
    int eventID;
    char eventName[32];
    int organiserID;
    int venueID;
    date eventDate;
    Time startTime;
    Time endTime;
    Time regDue;
    char* description;
} event;

// Linked list node for events
typedef struct EventNode {
    struct event evt;
    struct EventNode *next;
} EventNode;

// BST node for events by eventID
typedef struct EventBST {
    struct event evt;
    int height;
    struct EventBST *left;
    struct EventBST *right;
} EventBST;

```

```
// Function prototypes
void loadEvents(void);
void cleanPastEvents(void);
void viewEvents(void);
void sortEventByTime(void);
void sortEventChronological(void);
void sortEventByID(void);
event* searchEventID(void);
EventBST* newBSTNode(struct event e);
EventBST* insertBST(EventBST* root, struct event e);
EventBST* searchBST(EventBST* root, int eventID);
EventBST* minValueNode(EventBST* node);
EventBST* deleteBST(EventBST* root, int eventID);
date getCurrentDate(void);
int isPastEvent(struct date eventDate);
void quickSort(struct event arr[], int low, int high, int (*compare)(event,
    event));
int partition(struct event arr[], int low, int high, int (*compare)(event,
   event));
int compareChronological(event a, event b);
int compareByDate(event a, event b);
int compareByID(event a, event b);
int listToArray(event arr[]);
void addEvent(void);
void deleteEvent(void);
void modifyEvent(void);
void addToList(event e);
int checkValidTime(Time regEndtime, Time startTime, Time endTime);
int checkValidDate(date d);
int generateEventID(void);
void listEventsOfOrganizer();
void modifyEventDetailsInOrganizerFile(event modified);
int height(EventBST *n);
int maximum(int a, int b);
int getBalance(EventBST *n);
void calculateCost(event e);
EventBST *rightRotate(EventBST* y);
EventBST *leftRotate(EventBST *x);

#endif
```

Listing 2: Complete events.h header file (partial)

```c
#ifndef LOGIN_REGISTRATION_H
#define LOGIN_REGISTRATION_H

#include <stdbool.h>
#include <stdio.h>

typedef struct user {
    int userId;
    char name[64];
    unsigned int noOfEventsAttended;
    long long mobileNumber;
    char email[128];
} user;

typedef struct userStatus {
    int userId; // 0 if no user logged in and any number if user logged in
    char name[64];
    bool status; // true = logged in, false = not logged in
    bool isOrg; // if org = true, else false;
} userStatus;

int registerAsUser(int);
int loginAsUser(int);
int giveUserDetails(char *email, int id, FILE *fp, char *givenName);
int userValidation(user *a, int choice);
long long giveValidMobileNumber();
userStatus getDetails();
void logout();

#endif
```

Listing 3: Complete login_registration.h header file

```c
#ifndef ATTENDEE_H
#define ATTENDEE_H

#include <stdbool.h>
#include "login_registration.h"
// Attendee structure
typedef struct Attendee {
    int attendeeID;
    char name[64];
    char email[128];
    unsigned long long phoneNo;
    int eventID;
    int eventsRegistered;
    char status[15];           // present, absent or only registered
    char registrationDate[30];
} Attendee;

// Linked List Node
typedef struct Node {
    Attendee data;
    struct Node* next;
} Node;

#define DECREASE 1
#define INCREASE 2
// Attendee Functions (only attendees can call)
void registerAttendeeForEvent(Node** head, int eventID, userStatus *user);
bool unregisterAttendee(Node** head, userStatus* user);
bool fetchUserData(int userID, Attendee *a);
void updateEventsAttended(int userID, int choice);

// organizer functions
void markAttendance(Node** head);
void viewAllAttendees(Node* head, int eventID);
void viewStatistics(Node *head);

// common function
void getCurrentDateTime(char* buffer);
void searchAttendee(Node* head);
void saveToFile(Node* head, int eventID);
void loadFromFile(Node** head, int eventID);
void freeList(Node* head);

#endif
```

Listing 4: Complete Attendee.h header file (partial)

# 6 References


1. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Fundamentals of Computer Algorithms*, Universities Press, 2nd edition (2008).

2. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.

3. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.

4. TutorialsPoint. *Find out the current working directory in C/C++*. Retrieved from: current working directory functions

5. YouTube. *C Programming File Handling*. Retrieved from: file handling

6. GeeksforGeeks. *fseek() in C with example*. Retrieved from: `https://www.geeksforgeeks.org/cpp/fseek-in-c-with-example/`

7. Stack Overflow. *Does fscanf moves the passed file pointer ahead?* Retrieved from: stack Overflow reference

8. Stack Overflow. *How to check if a file has content or not using C?* Retrieved from: stack Overflow reference

9. GeeksforGeeks. *C - File Pointer*. Retrieved from: file pointers

10. GeeksforGeeks. *memset() in C with example*. Retrieved from: memset function

11. GeeksforGeeks. *fgets() and gets() in C language*. Retrieved from: fgets function

# 7 Conclusions

The Event Management System developed for this project demonstrates how classical data structures in C can be applied to a practical and meaningful real-world scenario. By employing linked lists for attendee management, AVL trees for fast event lookup, hash tables for venue indexing, and interval trees for scheduling conflict detection, the system combines efficiency with clarity.

The project provided hands-on experience with modular program design, file-based persistence, and algorithmic problem-solving. It also highlighted the importance of choosing appropriate data structures to meet specific operational requirements such as quick search, dynamic resizing, and conflict-free scheduling.

Although the system is intentionally simple, the architecture makes it easy to extend into a full-fledged management platform with better interfaces, database integration, and scalability. In its current form, the EMS succeeds in demonstrating fundamental concepts of software design and serves as a strong foundation for future academic or professional development.