GIK Institute of Eng. and Tech.

Faculty of Computer Sciences and Engineering

Data Structures and Algorithms, Term Project

(100 marks)

PROJECTS OVERVIEW TABLE:

S. No	Project Name	Project Core Functionalities	Required Functionalities	Data Structure Usage	Use Case / Scenario
1	**Smart Campus Bus Routing and Reservation System**	Manage campus bus stops, trips, and seat reservations	List stops; show departures/arrivals; reachability; shortest path; create/cancel reservations; passenger lists	Graph (stops & routes), hash table (stops/trips), arrays/lists, sorting, linked lists	Planning campus bus rides and managing seat bookings for students
2	**University Course Registration and Prerequisite Navigator**	Manage course catalog, prerequisites, and student enrollments	Show catalog; show prereqs; graph traversal; cycle detection; enroll/drop courses; timetables	Directed graph (prereqs), hash table (courses), BST (students), linked lists	Helping students register only in allowed courses and explore valid curriculum paths
3	**Library Catalog, Borrowing and Recommendation System**	Manage books, members, loans, and related-book graph	Search books; issue/return; due dates; related-book traversal; borrowers/history	BST (books by title), hash table (by ID), graph (recommendations), linked lists (loans)	Small library system with simple recommendations
4	**Hospital Patient Appointment and Referral Management System**	Manage doctors, patients, appointments, and department referrals	Avoid schedule conflicts; list schedules; referrals graph; shortest referral chain; stats	BST (doctors), hash table (patients), adjacency lists (departments graph), linked lists (appointments)	Outpatient appointment booking with referrals between departments
5	**Online Food Delivery and Rider Assignment System**	Manage restaurants, riders, orders, and city graph	Show restaurants; pending orders; assign riders; shortest delivery route; customer history	Graph (areas & roads), priority queue/heap (rider load), hash table (orders), lists	Food delivery platform backend with basic routing and assignment
6	**Ride-Sharing and Carpool Matching System**	Manage drivers, passengers, offers, requests, and routes	Register offers/requests; graph traversal; matching; shortest paths; popularity stats	Graph (road network), hash table (rides), priority queue (urgent requests), trees/lists	Matching drivers and passengers for shared rides

7	**Package Courier Routing and Tracking System**	Manage hubs, routes, shipments, and tracking	Register shipments; show neighbours; compute shipment paths; update status; hub statistics	Graph (hubs & routes), hash table (shipments), lists (history), simple arrays	Courier company routing and tracking parcels
8	**Cinema Ticket Booking and Seat Allocation System**	Manage movies, shows, halls, seats, and customers	Show movies/shows; seat maps; reserve/cancel seats; customer bookings; stats	2D arrays (seats), BST (movies), hash table (customers), lists (bookings), sorting	Multiplex cinema ticketing and seating
9	**Mini Social Network with Friend Recommendations**	Manage users, friendships, and posts	Add users/friendships; friend-of-friend traversal; shortest path; friend suggestions; posts	Graph (users & friendships), hash table (users), lists (posts), degree counting	Tiny social network with basic recommendation mechanics
10	**Smart Parking Allocation and Navigation System**	Manage parking zones, slots, and vehicles	Allocate/free slots; shortest path inside parking; occupancy stats; history	Graph (zones & paths), BST/heap (free slots), hash table (vehicles), lists	Multi-level parking lot slot management and navigation
11	**E-Commerce Product Catalog and Order Management System**	Manage products, categories, customers, and orders	Product search; category views; add/update products; place orders; best sellers	Tree (products by name), tree/graph (categories), hash table (orders), lists	Simple online store backend
12	**Metro/Train Network Planner and Ticket Reservation System**	Manage stations, trains, schedules, and tickets	Departures/arrivals; reachability; shortest paths; journeys with connections; bookings	Graph (station network), arrays/lists (trains), linked lists (tickets)	Metro/train route planning & reservations
13	**Event Management and Hall Timetabling System**	Manage halls, events, and conflicts	Create/allocate events; hall schedules; organiser views; conflict graph; stats	BST (halls, events), hash table (events), optional graph (conflicts)	Scheduling talks/workshops into halls without clashes
14	**Road Trip Planner with Points of Interest**	Manage cities, roads, POIs, and trips	Reachability; shortest path; POI lists; trips covering multiple POIs; stats	Graph (cities & roads), lists (POIs, trips), hash table (trips)	Planning road trips and itineraries
15	**File Backup and Folder Version History Manager**	Manage directory tree, backup versions, and file index	Directory tree; create/delete nodes; snapshots; shortest version path; restore; dedup	Tree (folders/files), graph (versions), hash table (file index), lists	Abstract backup/restore system with version history

Project 1: Smart Campus Bus Routing and Reservation System

Core focus: Trees, Graphs, Hashing and Sorting

Build a campus bus management tool. It maintains bus stops, trips between stops, and passenger reservations. The system should allow viewing the timetable, exploring which stops are reachable, finding shortest paths, and booking/cancelling seats on trips.

Required Functionalities

1. Display a list of all bus stops in tabular form. *(5 marks)*
2. For a selected stop, display all departures from that stop sorted by departure time. *(7 marks)*
3. For a selected stop, display all arrivals at that stop sorted by arrival time. *(7 marks)*
4. For a given stop, list all other stops reachable using one or more bus rides (BFS/DFS on the graph). *(9 marks)*
5. Show the sequence of stops on a minimum-time path between two stops (based on travel time only). *(10 marks)*
6. Compute the shortest travel time between two stops using Dijkstra (or similar) and print total minutes. *(10 marks)*
7. Create a reservation for a passenger on one or more bus trips and update seat counts. *(9 marks)*
8. Cancel an existing reservation and correctly adjust seat counts and booking lists. *(7 marks)*
9. For a given passenger, print all reservations sorted by travel date and time. *(8 marks)*
10. For a given trip, print the passenger list in alphabetical order of last name. *(8 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph (adjacency lists):** bus stops and routes between them.
- **Hash table / array:** indexing stops and trips by name/id.
- **Linked lists:** departures and arrivals per stop, booking lists.
- **Trees / sorting:** sorted passenger lists, time-ordered schedules.

Example Structures and Functions

```
struct BusTrip {
    int   tripId;
    char *startStop;
    char *endStop;
```

```
    int   departTime;       // hhmm
    int   arriveTime;       // hhmm
    int   freeSeats;

    BusTrip *nextFromStart; // next trip from same start stop
    BusTrip *nextToEnd;     // next trip arriving at same end stop
};

const int STOP_BUCKETS = 31;

struct StopEntry {
    char    *stopName;
    BusTrip *firstDeparture;   // sorted by departTime
    BusTrip *firstArrival;     // sorted by arriveTime
};

StopEntry stopTable[STOP_BUCKETS];

struct BusRoutePlan {
    int travelDay;             // mmdd
    int hopCount;
    int tripIds[10];           // sequence of tripIds
};

struct Booking {
    char    *firstName;
    char    *lastName;
    BusRoutePlan route;
    Booking *next;
};

Booking *bookingHead = nullptr;

BusTrip *CreateBusTrip(int tripId,
                       const char *startStop,
                       int departTime,
                       const char *endStop,
                       int arriveTime,
                       int freeSeats);

void InsertBusTrip(BusTrip *trip);
void ShowAllStops();
void ShowDepartures(const char *stopName);
void ShowArrivals(const char *stopName);

void ListReachableStops(const char *startStop);  // BFS/DFS
int  ShortestRoute(const char *fromStop,
                   const char *toStop,
                   BusRoutePlan &plan);

Booking *CreateBooking(const char *firstName,
                       const char *lastName,
                       const BusRoutePlan &plan);

void PrintBooking(const Booking *b);
void PrintPassengerList(int tripId);
void CancelBooking(const char *firstName,
                   const char *lastName,
```

```
                    int tripId);
```

Project 2: University Course Registration and Prerequisite Navigator

Core focus: Trees, Graphs, Hashing and Sorting

Build a course registration and curriculum planning tool. It maintains a catalog of courses, their prerequisites, and student registrations. The system should prevent students from enrolling in courses when prerequisites are not met and allow them to explore valid paths through the degree plan.

Required Functionalities

1. Display full course list (code, title, credits) in tabular form. *(5 marks)*
2. For a given course, show its direct prerequisites and all courses that depend on it. *(8 marks)*
3. Display all courses sorted by level and then by course code. *(7 marks)*
4. Starting from an intro course, show all reachable courses by repeatedly following prerequisite edges (BFS/DFS). *(9 marks)*
5. For a target advanced course, show one valid path of prerequisites from a starting basic course. *(10 marks)*
6. Detect and report cycles in the prerequisite graph (invalid curriculum). *(9 marks)*
7. Enrol a student in a course only if all prerequisites have been completed. *(9 marks)*
8. Drop a course from a student's current registration. *(6 marks)*
9. Print a student's timetable sorted by day and time. *(7 marks)*
10. For a given course, print list of enrolled students sorted by roll number or name. *(10 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Directed graph:** nodes = courses; edge `A → B` means *"A is a prerequisite of B"*.
- **Hash table:** fast course lookup by course code.
- **BST or tree:** students sorted by roll number; optional course ordering.
- **Linked lists:** course lists per student (completed / registered).

Example Structures and Functions

```
struct Course {
    char   *code;        // e.g., "CS2001"
```

```
    char   *title;
    int     creditHours;
    int     level;       // semester or level number

    Course **prereq;     // array of prerequisite pointers
    int     prereqCount;

    Course *nextByCode;  // for sorted list or BST
};

const int COURSE_TABLE = 53;

struct CourseBucket {
    char   *keyCode;
    Course *firstCourse;
};

CourseBucket courseTable[COURSE_TABLE];

/* Student side */

struct CourseRef {
    Course    *course;
    int        grade;      // -1 if currently taking
    CourseRef *next;
};

struct Student {
    int        rollNo;
    char      *firstName;
    char      *lastName;
    CourseRef *completedHead;
    CourseRef *registeredHead;
    Student   *left;       // BST by rollNo
    Student   *right;
};

Student *studentRoot = nullptr;

/* Prototypes */

Course *CreateCourse(const char *code,
                     const char *title,
                     int creditHours,
                     int level);

void InsertCourse(Course *c);
Course *FindCourse(const char *code);

void AddPrerequisite(const char *courseCode,
                     const char *prereqCode);

void ExploreCourseGraph(const char *startCode);
bool HasPrerequisiteCycle();

Student *CreateStudent(int rollNo,
                       const char *firstName,
                       const char *lastName);
```

```
bool EnrolStudentInCourse(int rollNo,
                          const char *courseCode);

bool DropStudentCourse(int rollNo,
                       const char *courseCode);

void PrintStudentTimetable(int rollNo);
void PrintCourseRoster(const char *courseCode);
```

Project 3: Library Catalog, Borrowing and Recommendation System

Core focus: Trees, Graphs, Hashing and Sorting

Implement a small library backend that manages books, members, borrowing activity, and "related book" links. Users should be able to search the catalog, issue and return books, and explore related titles via a recommendation graph.

Required Functionalities

1. Display all books (id, title, author, subject, availability). *(5 marks)*
2. Search for books by id or title prefix; show matches sorted by title. *(8 marks)*
3. For a given subject/category, list all books sorted by title or author. *(7 marks)*
4. For a book, display all related books using graph traversal (e.g., BFS on neighbours). *(9 marks)*
5. For two books A and B, print one path of related books connecting them (if exists). *(10 marks)*
6. Issue a book to a member, update availability and record due date. *(10 marks)*
7. Return a book, update the record, and optionally compute a simple fine. *(9 marks)*
8. For a member, print all currently issued books sorted by due date. *(7 marks)*
9. For a given book, print the list of members who have borrowed it, sorted by member ID or last name. *(8 marks)*
10. Show basic statistics such as most issued books or top borrowers. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **BST by title:** sorted catalog views.
- **Hash table by book id:** quick lookups.
- **Graph:** related books as an undirected recommendation graph.
- **Linked lists:** loan history per member; borrower lists per book.

Example Structures and Functions

```
struct Book {
    int     bookId;
    char   *title;
    char   *author;
    char   *subject;
    int     inStock;

    Book   *left;      // BST by title
    Book   *right;

    Book  **neighbours;
    int     neighbourCount;
};

const int BOOK_BUCKETS = 101;

struct BookBucket {
    int   keyId;
    Book *firstBook;
};

BookBucket bookTable[BOOK_BUCKETS];

struct Loan {
    int   bookId;
    int   issueDate;   // yyyymmdd
    int   dueDate;
    int   returnDate;  // 0 if not returned
    Loan *next;
};

struct Member {
    int    memberId;
    char  *firstName;
    char  *lastName;
    Loan  *loanHead;
    Member *left;      // BST by id
    Member *right;
};

Member *memberRoot = nullptr;

Book *CreateBook(int id,
                 const char *title,
                 const char *author,
                 const char *subject,
                 int copies);

void InsertBook(Book *b);
Book *FindBookById(int id);

void LinkRelatedBooks(int idA, int idB);
void ShowRelatedBooks(int id);

Member *CreateMember(int memberId,
                     const char *firstName,
```

```
                    const char *lastName);

bool IssueBook(int memberId,
               int bookId,
               int issueDate,
               int dueDate);

bool ReturnBook(int memberId,
                int bookId,
                int returnDate);

void PrintCurrentLoans(int memberId);
void PrintBookBorrowers(int bookId);
```

Project 4: Hospital Patient Appointment and Referral Management System

Core focus: Trees, Graphs, Hashing and Sorting

Model a simple outpatient clinic. The system manages doctors, patients, time-based appointments, and referrals between hospital departments. It should avoid scheduling clashes and allow exploration of referral chains.

Required Functionalities

1. Display all doctors (id, name, specialty, department) in tabular form. *(5 marks)*
2. For a given doctor, show appointments for a specific day sorted by start time. *(8 marks)*
3. For a given patient, list all upcoming appointments sorted by date and time. *(7 marks)*
4. Insert a new appointment only if it does not overlap with existing appointments of that doctor. *(10 marks)*
5. Cancel an existing appointment and update all relevant lists. *(9 marks)*
6. Represent departments and referrals as a directed graph and, for a department, list all departments reachable by referrals. *(9 marks)*
7. For two departments A and B, find a shortest referral chain (fewest edges) and print the sequence. *(10 marks)*
8. For a given doctor, print the patient list in alphabetical order (use tree/sorting). *(7 marks)*
9. Show simple statistics such as the doctor with the highest number of appointments and the busiest department. *(8 marks)*
10. Save all doctors, patients, appointments, and referrals to text files on exit. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **BST:** doctors sorted by id.
- **Linked lists:** appointments per doctor, per patient.
- **Graph (adjacency lists):** referral links between departments.
- **Hash table:** fast lookup of patients by ID.

Example Structures and Functions

```
struct Appointment {
    int   appId;
    int   date;       // yyyymmdd
    int   startTime;  // hhmm
    int   endTime;    // hhmm
    int   patientId;

    Appointment *next; // sorted by date+startTime
};

struct Doctor {
    int          doctorId;
    char        *name;
    char        *specialty;
    char        *departmentName;
    Appointment *appointments;
    Doctor      *left;   // BST by id
    Doctor      *right;
};

Doctor *doctorRoot = nullptr;

/* Departments graph */

struct DeptNode;

struct DeptEdge {
    DeptNode *to;
    DeptEdge *next;
};

struct DeptNode {
    char     *name;
    DeptEdge *firstEdge;
    DeptNode *next;
};

DeptNode *deptHead = nullptr;

/* Patients hash table */

const int PATIENT_BUCKETS = 41;

struct Patient {
    int      patientId;
    char    *firstName;
    char    *lastName;
```

```
    Patient *next;
};

Patient *patientTable[PATIENT_BUCKETS];

Doctor *CreateDoctor(int id,
                     const char *name,
                     const char *spec,
                     const char *dept);

bool AddAppointment(int doctorId,
                    int appId,
                    int date,
                    int startTime,
                    int endTime,
                    int patientId);

bool CancelAppointment(int appId);

void PrintDoctorSchedule(int doctorId,
                         int date);

DeptNode *GetOrCreateDept(const char *name);
void AddReferral(const char *fromDept,
                 const char *toDept);

void ListReachableDepartments(const char *startDept);
int  ShortestReferralPath(const char *fromDept,
                          const char *toDept);
```

Project 5: Online Food Delivery and Rider Assignment System

Core focus: Graphs, Hashing, Priority Queues and Sorting

Create a simplified back-end for a food delivery platform. The system handles restaurants, customers, riders and delivery orders. It must compute delivery routes and assign orders to riders efficiently.

Required Functionalities

1. Display a table of all restaurants (id, name, area). *(5 marks)*
2. Show all pending orders sorted by creation time. *(8 marks)*
3. For a selected rider, display their active orders sorted by delivery deadline or distance. *(7 marks)*
4. Represent the city as a weighted graph of areas; from a restaurant, list all reachable areas via BFS/DFS. *(9 marks)*
5. For a given order, compute the shortest path from restaurant to customer and display the route. *(10 marks)*
6. Assign an order to the “best” rider (e.g., minimal load or closest area) using a priority structure. *(10 marks)*

7. Maintain a hash table of orders for fast lookup by order id and support status updates (placed/assigned/delivered). *(9 marks)*
8. For a given customer, print all past orders in chronological order. *(7 marks)*
9. Print a ranking of riders based on the number of completed deliveries. *(8 marks)*
10. Save all restaurants, riders and orders to disk when the program exits. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** areas and roads between them.
- **Priority queue / heap:** choose best rider for next order.
- **Hash table:** orders by id.
- **Linked lists / arrays:** lists of orders, riders, customers.

Example Structures and Functions

```
struct Area;

struct Road {
    Area  *to;
    int    distance;   // or time
    Road  *next;
};

struct Area {
    char  *name;
    Road  *firstRoad;
    Area  *next;
};

Area *areaHead = nullptr;

struct Restaurant {
    int        restId;
    char      *name;
    Area      *location;
    Restaurant *next;
};

struct Rider {
    int      riderId;
    char    *name;
    Area    *currentArea;
    int      activeLoad;
    Rider   *left;    // for BST or heap
    Rider   *right;
};

struct Order {
```

```
    int        orderId;
    int        timeCreated;
    int        deadline;
    Restaurant *fromRest;
    char      *customerName;
    Area      *customerArea;
    int        status;       // 0=pending,1=assigned,2=delivered
    Rider     *assignedRider;
    Order     *next;
};

const int ORDER_BUCKETS = 97;
Order *orderTable[ORDER_BUCKETS];

int   OrderHash(int orderId);
Area *GetOrCreateArea(const char *name);
Restaurant *CreateRestaurant(int id,
                             const char *name,
                             const char *areaName);
Rider *CreateRider(int riderId,
                   const char *name,
                   const char *startArea);

Order *CreateOrder(int orderId,
                   int timeCreated,
                   int deadline,
                   int restId,
                   const char *custName,
                   const char *custArea);

int  AssignOrderToRider(int orderId);
void PrintCustomerOrders(const char *customerName);
void PrintRiderRanking();
```

Project 6: Ride-Sharing and Carpool Matching System

Core focus: Graphs, Hashing, Priority Queues and Trees

Implement a ride-sharing platform where drivers advertise routes and passengers post ride requests. The system should match compatible drivers and passengers and group them into carpools based on the road network.

Required Functionalities

1. Display all registered drivers (id, name, home area, capacity, rating). *(5 marks)*
2. Register a new ride offer from a driver (origin, destination, departure time, seats). *(8 marks)*
3. Register a new ride request from a passenger (origin, destination, earliest departure, latest arrival). *(7 marks)*
4. Represent roads as a graph and, for each offer, show areas reachable within a given cost bound. *(9 marks)*

5. For a given driver and passenger, determine if the passenger's path can be embedded into the driver's route and show a shared path. *(10 marks)*
6. Maintain a priority queue of pending requests (e.g., earliest departure first) and always match the most urgent request first. *(10 marks)*
7. Maintain a hash table of active rides; for a ride id, print all details and passengers in that car. *(9 marks)*
8. Print ride history for a given user (driver or passenger) sorted by time. *(7 marks)*
9. Print top-k drivers with most completed rides. *(8 marks)*
10. Save all users, offers, requests and rides to files at exit. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** road network.
- **Hash table:** rides, users.
- **Priority queue / heap:** pending requests.
- **BST / lists:** driver ranking, history.

Example Structures and Functions

```
struct Place;

struct RoadLink {
    Place    *to;
    int       cost;
    RoadLink *next;
};

struct Place {
    char     *name;
    RoadLink *firstLink;
    Place    *next;
};

Place *placeHead = nullptr;

struct User {
    int     userId;
    char   *name;
    int     isDriver;   // 1=driver,0=passenger
    int     rating;
    User   *left;       // BST
    User   *right;
};

struct RideOffer {
    int     offerId;
```

```
    int     driverId;
    Place  *startPlace;
    Place  *endPlace;
    int     departTime;
    int     capacity;
    int     seatsLeft;
    RideOffer *next;
};

struct RideRequest {
    int     requestId;
    int     passengerId;
    Place  *fromPlace;
    Place  *toPlace;
    int     earliest;
    int     latest;
    RideRequest *next;
};

User *userRoot = nullptr;
RideOffer  *offerHead  = nullptr;
RideRequest *requestHead = nullptr;

Place *GetOrCreatePlace(const char *name);
void   AddRoad(const char *from,
               const char *to,
               int cost);

User *CreateUser(int userId,
                 const char *name,
                 int isDriver);

RideOffer  *CreateRideOffer(int offerId, int driverId,
                            const char *start, const char *end,
                            int departTime, int capacity);

RideRequest *CreateRideRequest(int requestId, int passengerId,
                               const char *from, const char *to,
                               int earliest, int latest);

int  MatchNextRequest();
void PrintUserHistory(int userId);
void PrintTopDrivers(int k);
```

Project 7: Package Courier Routing and Tracking System

Core focus: Graphs, Hashing, Lists and Sorting

Simulate a courier company that ships packages between hubs. The system keeps track of hubs, routes, shipments and their statuses, and it computes routes between hubs.

Required Functionalities

1. Display all hubs (id, city, country) in tabular form. *(5 marks)*
2. Register a new shipment with origin hub, destination hub, weight and creation date. *(8 marks)*
3. Display all active (not delivered) shipments sorted by creation date. *(7 marks)*
4. For a hub, list all directly connected hubs (neighbours) with route distances. *(8 marks)*
5. For a shipment, compute a shortest path from origin to destination and show the hub sequence. *(10 marks)*
6. Update the status of a shipment when it moves to next hub and record the travel history. *(10 marks)*
7. Maintain a hash table from tracking number to shipment, and support direct lookup of full status. *(10 marks)*
8. For a hub, list all shipments currently passing through it sorted by arrival time or id. *(7 marks)*
9. Show statistics such as the hub with the largest number of incoming shipments. *(8 marks)*
10. Save the route network and all shipments to files on exit. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** hubs and directed/undirected routes.
- **Hash table:** shipments by tracking number.
- **Linked lists:** shipments per hub and global lists.
- **Sorting / arrays:** time-sorted views.

Example Structures and Functions

```
struct Hub;

struct RouteArc {
    Hub      *to;
    int       distance;
    RouteArc *next;
};

struct Hub {
    int      hubId;
    char    *city;
    char    *country;
    RouteArc *firstArc;
    Hub     *next;
};

Hub *hubHead = nullptr;

struct Shipment {
    int     trackingNo;
```

```
    Hub    *origin;
    Hub    *destination;
    int     weightKg;
    int     dateCreated;
    int     status;        // 0=in-transit,1=delivered

    Hub   **path;          // planned path of hubs
    int     pathLen;
    int     currentIndex;

    Shipment *nextByDate;
    Shipment *nextAtHub;
};

const int SHIP_BUCKETS = 101;
Shipment *shipmentTable[SHIP_BUCKETS];

Hub *GetOrCreateHub(int hubId,
                    const char *city,
                    const char *country);

void  AddRoute(int fromHubId,
               int toHubId,
               int distance);

void  PrintNeighbours(int hubId);
int   ComputeShipmentPath(int fromHubId,
                          int toHubId,
                          Hub *path[],
                          int &len);

Shipment *CreateShipment(int trackingNo,
                         int fromHubId,
                         int toHubId,
                         int weightKg,
                         int dateCreated);

bool AdvanceShipment(int trackingNo);
bool MarkDelivered(int trackingNo);

void PrintShipment(int trackingNo);
void PrintShipmentsAtHub(int hubId);
```

Project 8: Cinema Ticket Booking and Seat Allocation System

Core focus: Arrays, Hashing, Trees and Sorting

Model a multi-screen cinema. The system manages movies, shows, seating and customer bookings. It should allow viewing schedules, reserving seats, cancelling reservations and listing customers for each show.

Required Functionalities

1. Display all movies (id, title, rating) in tabular form. *(5 marks)*

2. For a selected movie, list all shows (screen, date, time) sorted by date/time. *(8 marks)*
3. For a show, display the seating layout and highlight available vs. reserved seats. *(7 marks)*
4. Reserve one or more seats for a customer in a given show (if free) and update seat map. *(10 marks)*
5. Cancel a customer's reservation and free the corresponding seats. *(9 marks)*
6. For a show, print the customer list sorted by last name. *(9 marks)*
7. Maintain a hash table of customers for quick lookup of their bookings. *(8 marks)*
8. For a customer id, print all future bookings sorted by date and time. *(7 marks)*
9. Generate simple statistics: tickets sold per show and per movie. *(7 marks)*
10. Save all shows and reservations to files before program exit. *(10 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **2D array (flattened):** seat maps per show.
- **BST:** movies sorted by title.
- **Hash table:** customers by id.
- **Linked lists:** bookings per customer, per show.

Example Structures and Functions

```
struct Movie {
    int     movieId;
    char   *title;
    char   *rating;
    Movie  *left;     // BST by title
    Movie  *right;
};

struct Show {
    int     showId;
    Movie  *movie;
    char   *screenName;
    int     date;     // yyyymmdd
    int     time;     // hhmm
    int     rows;
    int     cols;
    int    *seats;    // 0=free,1=reserved (rows*cols)

    Show   *nextForMovie;
};

struct BookingEntry {
    int           showId;
    int           row;
    int           col;
    BookingEntry *next;
```

```
};

struct Customer {
    int         customerId;
    char       *firstName;
    char       *lastName;
    BookingEntry *bookings;
    Customer   *next;
};

const int CUSTOMER_BUCKETS = 97;
Customer *customerTable[CUSTOMER_BUCKETS];

Movie *CreateMovie(int movieId,
                   const char *title,
                   const char *rating);

Show  *CreateShow(int showId,
                  int movieId,
                  const char *screen,
                  int date,
                  int time,
                  int rows,
                  int cols);

int   ReserveSeat(int showId,
                  int customerId,
                  int row,
                  int col);

int   CancelSeat(int showId,
                 int customerId,
                 int row,
                 int col);

Customer *GetOrCreateCustomer(int customerId,
                              const char *firstName,
                              const char *lastName);

void  PrintSeatMap(int showId);
void  PrintCustomerBookings(int customerId);
void  PrintShowCustomers(int showId);
```

Project 9: Mini Social Network with Friend Recommendations

Core focus: Graphs, Hashing, Lists and Simple Ranking

Implement a mini social network where users can befriend each other, post small status updates, and receive friend recommendations based on mutual connections.

Required Functionalities

1. Display all users (id, name, city) in tabular form. *(5 marks)*

2. Create a new user and insert into the network. *(8 marks)*
3. Add a friendship link between two users (undirected edge). *(7 marks)*
4. For a user, list direct friends and friends-of-friends using BFS/DFS. *(9 marks)*
5. For two users A and B, print one shortest friendship path between them. *(10 marks)*
6. Suggest new friends for a user based on the number of mutual friends (top-k recommendations). *(10 marks)*
7. Allow a user to create a short text post, stored in their timeline in reverse chronological order. *(9 marks)*
8. Print the timeline of a user (newest post first). *(7 marks)*
9. Show a ranking of the most "popular" users (highest degree in graph). *(7 marks)*
10. Save all users, friendships and posts to files at program termination. *(8 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** adjacency lists for friendships.
- **Hash table:** user lookup by id.
- **Linked lists:** posts per user (timeline).
- **Counting / sorting:** popularity ranking.

Example Structures and Functions

```
struct User;

struct FriendLink {
    User       *to;
    FriendLink *next;
};

struct Post {
    int    postId;
    int    timeStamp;
    char  *text;
    Post  *next;
};

struct User {
    int        userId;
    char      *name;
    char      *city;
    FriendLink *firstFriend;
    Post      *timeline;
    User      *next;       // hash bucket list
};

const int USER_BUCKETS = 101;
```

```
User *userTable[USER_BUCKETS];

int   UserHash(int userId);
User *CreateUser(int userId,
                 const char *name,
                 const char *city);

bool  AddFriendship(int userIdA,
                    int userIdB);

void  ShowFriendNetwork(int userId);
int   ShortestFriendPath(int fromId,
                         int toId,
                         int path[],
                         int &len);

bool  AddPost(int userId,
              int postId,
              int timeStamp,
              const char *text);

void  PrintTimeline(int userId);
void  PrintTopPopularUsers(int k);
```

Project 10: Smart Parking Allocation and Navigation System

Core focus: Graphs, Trees/Heaps and Hashing

Simulate a multi-level parking lot. The system should allocate slots to arriving vehicles, support navigation to slots, track departures, and provide occupancy statistics.

Required Functionalities

1. Show an overview of all parking zones (level, zone id, capacity, free spaces). *(5 marks)*
2. When a car arrives, allocate the “best” available slot using a priority approach. *(9 marks)*
3. Mark a car as departed and free its slot. *(8 marks)*
4. Model paths between zones as a graph; for a given entrance and slot, display a shortest route. *(10 marks)*
5. For a vehicle registration number, print where the car is parked (level, zone, slot). *(9 marks)*
6. Maintain a history of parked vehicles and their duration of stay. *(8 marks)*
7. For a level, print all occupied slots sorted by slot id. *(7 marks)*
8. Show statistics such as peak occupancy and most used level. *(7 marks)*
9. Save the current parking state to text files and restore it on startup. *(7 marks)*
10. Provide a simple search menu to find cars or free spaces. *(10 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** zones and paths for navigation.
- **Tree / heap:** structure of free slots by “priority”.
- **Hash table:** vehicles by registration number.
- **Lists:** history records, occupancy lists.

Example Structures and Functions

```
struct Zone;

struct PathEdge {
    Zone     *to;
    int       distance;
    PathEdge *next;
};

struct Zone {
    int       level;
    int       zoneId;
    int       totalSlots;
    int       freeSlots;
    PathEdge *firstPath;
    Zone     *next;
};

Zone *zoneHead = nullptr;

struct Slot {
    int    level;
    int    zoneId;
    int    slotNo;
    int    isFree;
    Slot  *left;   // BST or heap-like
    Slot  *right;
};

Slot *slotRoot = nullptr;

struct Vehicle {
    char    *regNo;
    int      level;
    int      zoneId;
    int      slotNo;
    int      timeIn;
    int      timeOut;
    Vehicle *next;
};

const int VEH_BUCKETS = 101;
Vehicle *vehicleTable[VEH_BUCKETS];

int     VehicleHash(const char *regNo);
Zone   *CreateZone(int level,
```

```
                    int zoneId,
                    int totalSlots);
void    AddZonePath(int levelA, int zoneA,
                    int levelB, int zoneB,
                    int distance);
void    InitialiseSlots();
bool    AllocateSlot(const char *regNo,
                     int currentTime);
bool    FreeSlot(const char *regNo,
                 int departTime);
void    PrintLevelOccupancy(int level);
```

Project 11: E-Commerce Product Catalog and Order Management System

Core focus: Trees, Hashing, Lists and Sorting

Create a simple back-end for an online store. The system maintains product catalog, categories, customers and orders. It must support product search, order placement and sales statistics.

Required Functionalities

1. Display all products (id, name, price, stock, category) in tabular form. *(5 marks)*
2. Search for products by name prefix and display results sorted by name. *(8 marks)*
3. For a selected category, list all products sorted by price. *(7 marks)*
4. Insert new products and update stock quantities. *(9 marks)*
5. Create new customers and store their details. *(8 marks)*
6. Place an order with multiple items; update stock and store order details. *(10 marks)*
7. Maintain a hash table of orders for quick lookup by order id. *(9 marks)*
8. For a customer, print all orders in chronological order with totals. *(7 marks)*
9. Rank products by number of times ordered and show the top-k products. *(8 marks)*
10. Save products, customers and orders to disk at exit. *(9 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **BST:** products sorted by name.
- **Tree/graph:** category hierarchy (parent/child).
- **Hash table:** orders by id.
- **Linked lists:** items per order, orders per customer.

Example Structures and Functions

```
struct Category {
    int        catId;
    char      *name;
    Category  *parent;
    Category  *firstChild;
    Category  *nextSibling;
};

struct Product {
    int       productId;
    char     *name;
    double    price;
    int       stock;
    Category *category;
    int       orderCount;

    Product  *left;   // BST by name
    Product  *right;
};

struct OrderItem {
    int        productId;
    int        quantity;
    double     unitPrice;
    OrderItem *next;
};

struct Order {
    int        orderId;
    int        customerId;
    int        date;    // yyyymmdd
    OrderItem *items;
    double     total;
    Order     *nextForCustomer;
    Order     *next;
};

struct Customer {
    int        customerId;
    char      *firstName;
    char      *lastName;
    Order     *orders;
    Customer  *next;
};

const int ORDER_BUCKETS = 101;
Order    *orderTable[ORDER_BUCKETS];

Product  *CreateProduct(int id,
                        const char *name,
                        double price,
                        int stock,
                        int catId);

Customer *CreateCustomer(int id,
                         const char *firstName,
```

```
                          const char *lastName);

Order    *PlaceOrder(int orderId,
                     int customerId,
                     int date);

int       OrderHash(int orderId);
Order    *FindOrder(int orderId);
void      PrintCustomerOrders(int customerId);
void      PrintTopProducts(int k);
```

Project 12: Metro/Train Network Planner and Ticket Reservation System

Core focus: Graphs, Arrays and Linked Lists

Model a metro or train network with stations, trains, schedules and bookings. The system must show departures/arrivals, support route planning and manage ticket reservations.

Required Functionalities

1. Show all stations in the network. *(5 marks)*
2. For a station, list all departing trains sorted by departure time. *(8 marks)*
3. For a station, list all arriving trains sorted by arrival time. *(7 marks)*
4. From a starting station, display all reachable stations using BFS/DFS on the station graph. *(9 marks)*
5. For two stations A and B, print a shortest-time station path based on in-train travel times only. *(10 marks)*
6. Plan a journey (possibly multi-leg) that respects minimum connection times between trains. *(10 marks)*
7. Create a ticket booking for a passenger and update seat counts of involved trains. *(9 marks)*
8. Cancel a booking and free seats on all trains in the journey. *(7 marks)*
9. For a train id, print the passenger manifest sorted by last name. *(8 marks)*
10. Save station graph, train schedules and bookings to files on exit. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** stations and direct rail connections.
- **Arrays / lists:** trains.
- **Linked lists:** tickets and journeys.
- **Sorting:** departures/arrivals, passenger manifests.

Example Structures and Functions

```
struct Station;

struct RailEdge {
    Station  *to;
    int       travelMinutes;
    RailEdge *next;
};

struct Station {
    char     *name;
    RailEdge *firstEdge;
    Station  *next;
};

Station *stationHead = nullptr;

struct Train {
    int       trainId;
    Station  *fromStation;
    Station  *toStation;
    int       departTime;
    int       arriveTime;
    int       totalSeats;
    int       bookedSeats;
    Train    *nextFrom;
    Train    *nextTo;
};

const int MAX_TRAINS = 128;
Train    *trainArray[MAX_TRAINS];
int       trainCount = 0;

struct JourneyPlan {
    int  day;               // mmdd
    int  legCount;
    int  trainIds[10];
};

struct Ticket {
    char        *firstName;
    char        *lastName;
    JourneyPlan  journey;
    Ticket      *next;
};

Ticket *ticketHead = nullptr;

Station *GetOrCreateStation(const char *name);
void     AddRail(const char *from,
                 const char *to,
                 int minutes);

Train   *CreateTrain(int id,
                     const char *from,
                     const char *to,
                     int departTime,
```

```
                        int arriveTime,
                        int totalSeats);

void     PrintDepartures(const char *stationName);
void     PrintArrivals(const char *stationName);

int      ShortestStationPath(const char *start,
                             const char *end,
                             Station *path[],
                             int &len);

int      PlanJourney(const char *start,
                     const char *end,
                     JourneyPlan &plan);

Ticket  *CreateTicket(const char *firstName,
                      const char *lastName,
                      const JourneyPlan &plan);

int      CancelTicket(const char *firstName,
                      const char *lastName);
```

Project 13: Event Management and Hall Timetabling System

Core focus: Trees, Hashing and Optional Graphs

Manage halls and events such as talks or workshops. The system should schedule events without conflicts, allow viewing timetables, and report basic statistics.

Required Functionalities

1. Print all halls (id, name, capacity, location) in tabular form. *(5 marks)*
2. Create a new event request with desired time interval, hall preferences and expected audience. *(8 marks)*
3. Allocate a hall to an event if there is no time clash and capacity is sufficient. *(9 marks)*
4. Display the schedule of a hall for a given day sorted by start time. *(9 marks)*
5. For a given organiser, list all events they are responsible for sorted by date/time. *(8 marks)*
6. Cancel an event and free the associated hall and time slot. *(9 marks)*
7. Maintain a hash table of events for direct lookup by event id and show event details. *(8 marks)*
8. Optionally, represent overlapping events as edges in a conflict graph and display conflicts for a given day. *(7 marks)*
9. Provide simple usage stats: hall usage counts, busiest day, most requested hall. *(7 marks)*
10. Save all halls and events to text files at exit. *(10 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **BST:** halls; events sorted by (date, start time).
- **Hash table:** events lookup by id.
- **Optional graph:** conflict graph between overlapping events.

Example Structures and Functions

```
struct Hall {
    int    hallId;
    char  *name;
    int    capacity;
    char  *location;
    Hall  *left;    // BST
    Hall  *right;
};

Hall *hallRoot = nullptr;

struct Event {
    int    eventId;
    char  *title;
    char  *organiser;
    int    date;       // yyyymmdd
    int    startTime;
    int    endTime;
    int    expected;
    int    hallId;     // -1 if unassigned
    Event *left;       // BST by (date, startTime)
    Event *right;
};

const int EVENT_BUCKETS = 101;
Event *eventTable[EVENT_BUCKETS];

int    EventHash(int eventId);
Hall  *CreateHall(int id,
                  const char *name,
                  int capacity,
                  const char *location);

Event *CreateEvent(int eventId,
                   const char *title,
                   const char *organiser,
                   int date,
                   int startTime,
                   int endTime,
                   int expected);

int    AllocateHall(int eventId);
void   PrintHallSchedule(int hallId,
                         int date);
Event *FindEvent(int eventId);
void   PrintHallStatistics();
```

Project 14: Road Trip Planner with Points of Interest

Core focus: Graphs, Lists and Hashing

Implement a route planner for road trips. The system manages cities, roads between them, and points of interest (POIs). Users can plan trips visiting specific POIs and view statistics on saved trips.

Required Functionalities

1. Show a list of all cities in the network with basic info. *(5 marks)*
2. Add a new road segment between two cities (distance and travel time). *(8 marks)*
3. From a starting city, list all reachable cities and cumulative distances using BFS/DFS. *(9 marks)*
4. For two cities A and B, compute and display the shortest path by distance. *(10 marks)*
5. Attach points of interest to cities and list all POIs for a city sorted by name. *(9 marks)*
6. Allow the user to specify a trip that must visit a set of POIs and generate a simple route that covers them (heuristic acceptable). *(10 marks)*
7. Maintain a set of saved trips, each with sequence of cities and total distance. *(8 marks)*
8. For a saved trip, print the itinerary in order. *(7 marks)*
9. Show statistics such as the longest saved trip and the most frequently visited city. *(7 marks)*
10. Save the road network, POIs and saved trips to text files. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Graph:** cities and roads.
- **Lists:** POIs per city; cities per trip.
- **Hash table:** trips by id.

Example Structures and Functions

```
struct City;

struct RoadEdge {
    City     *to;
    int       distanceKm;
    int       timeMinutes;
    RoadEdge *next;
};

struct City {
```

```
    char     *name;
    char     *country;
    RoadEdge *firstRoad;
    City     *next;
};

City *cityHead = nullptr;

struct Poi {
    char *name;
    char *type;
    Poi  *next;
};

struct CityPoiList {
    City        *city;
    Poi         *poiHead;
    CityPoiList *next;
};

CityPoiList *poiHead = nullptr;

struct Trip {
    int    tripId;
    City **citySeq;
    int    cityCount;
    int    totalDistance;
    Trip  *next;
};

const int TRIP_BUCKETS = 97;
Trip *tripTable[TRIP_BUCKETS];

City *GetOrCreateCity(const char *name,
                      const char *country);

void  AddRoad(const char *from,
              const char *to,
              int distKm,
              int timeMinutes);

void  ExploreFromCity(const char *name);
int   ShortestCityPath(const char *from,
                       const char *to,
                       City *path[],
                       int &len);

void  AddPoi(const char *cityName,
             const char *poiName,
             const char *type);

Trip *CreateTrip(int tripId,
                 City *citySeq[],
                 int cityCount);

int   TripHash(int tripId);
void  PrintTrip(int tripId);
```

Project 15: File Backup and Folder Version History Manager

Core focus: Trees, Graphs and Hashing

Abstract a file system and its backup history. The system keeps a directory tree for folders/files and a version graph for snapshots. It should support creating snapshots, restoring files and exploring version evolution.

Required Functionalities

1. Display the directory tree from a given folder using indentation (pre-order traversal). *(5 marks)*
2. Create a new file or folder in the tree under a specified parent path. *(8 marks)*
3. Delete a file or an empty folder from the tree. *(7 marks)*
4. Create a new snapshot (backup version) of the current state and link it to the previous snapshot. *(9 marks)*
5. For two versions A and B, display a shortest path of snapshots connecting them in the version graph. *(10 marks)*
6. Maintain a hash table that maps each file path to its metadata in the latest snapshot (size, hash, etc.). *(10 marks)*
7. Restore a particular file from a selected snapshot. *(9 marks)*
8. List all snapshots sorted by creation time with basic statistics (number of files, total size). *(7 marks)*
9. Detect duplicate files (same content hash) and report potential space saving by deduplication. *(8 marks)*
10. Save the directory tree and version graph to files on exit and reload them at startup. *(7 marks)*
11. Viva / quiz / code walk-through. *(20 marks)*

Data Structure Usage

- **Tree:** directory hierarchy (folders & files).
- **Graph:** snapshots and links between versions.
- **Hash table:** file index by full path or content hash.

Example Structures and Functions

```
struct FileNode {
    char      *name;
    int        isFolder;     // 1=folder,0=file
    int        sizeBytes;
    char      *contentHash;  // for dedup detection

    FileNode  *parent;
```

```
    FileNode  *firstChild;
    FileNode  *nextSibling;
};

FileNode *rootFolder = nullptr;

struct Version;

struct VersionEdge {
    Version    *to;
    VersionEdge *next;
};

struct Version {
    int        versionId;
    int        createTime;   // timestamp or yyyymmdd
    VersionEdge *firstEdge;
    Version    *next;
};

Version *versionHead = nullptr;

const int FILE_BUCKETS = 101;

struct FileIndexEntry {
    char           *fullPath;   // "/docs/report.txt"
    FileNode       *node;
    FileIndexEntry *next;
};

FileIndexEntry *fileIndex[FILE_BUCKETS];

FileNode *CreateFileNode(const char *name,
                         int isFolder,
                         int sizeBytes,
                         const char *hash);

int       InsertIntoFolder(FileNode *parent,
                           FileNode *child);

int       RemoveNode(FileNode *node);

int       FileHash(const char *path);
void      InsertIntoFileIndex(const char *path,
                              FileNode *node);

FileNode *FindFileByPath(const char *path);

Version  *CreateVersion(int versionId,
                        int createTime,
                        int previousVersionId);

int       ShortestVersionPath(int fromId,
                              int toId,
                              Version *path[],
                              int &len);

void      ReportDuplicates();
```