DAYANANDA SAGAR ACADEMY OF TECHNOLOGY AND MANAGEMENT

Udayapura, Kanakapura Road, Opp. Art of Living, Bangalore – 560082

(Affliated to VTU, Belagavi, Approved by AICTE, New Delhi)

Accredited by NBA and NAAC (A+)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DATA STRUCTURES & APPLICATIONS LAB MANUAL (BCS302)

Compiled by:

Prof. Yamini G

Dr. Nandini C
HOD, CSE, DSATM

Dr. M Ravishankar
Principal, DSATM

INSTITUTION VISION AND MISSION

Vision of the Institution

To strive at creating the institution a center of highest caliber of learning, so as to create an overall intellectual atmosphere with each deriving strength from the other to be the best of engineers, scientists with management &design skills.

Mission of the Institution:

- To serve its region, state, the nation and globally by preparing students to make
- meaningful contributions in an increasing complex global society challenge.
- To encourage, reflection on and evaluation of emerging needs and priorities with state of art infrastructure at institution.
- To support research and services establishing enhancements in technical, health, economic, human and cultural development.
- To establish inter disciplinary center of excellence, supporting/ promoting student's implementation.
- To increase the number of Doctorate holders to promote research culture on campus.
- To establish IIPC, IPR, EDC, innovation cells with functional MOU's supporting student's quality growth.

QUALITY POLICY

Dayananda Sagar Academy of Technology and Management aims at achieving academic excellence through continuous improvement in all spheres of Technical and Management education. In pursuit of excellence cutting-edge and contemporary skills are imparted to the utmost satisfaction of the students and the concerned stakeholders

OBJECTIVES & GOALS

- Creating an academic environment to nurture and develop competent entrepreneurs, leaders and professionals who are socially sensitive and environmentally conscious.
- Integration of Outcome Based Education and cognitive teaching and learning strategies to enhance learning effectiveness.
- Developing necessary infrastructure to cater to the changing needs of Business and Society.
- Optimum utilization of the infrastructure and resources to achieve excellence in all areas of relevance.
- Adopting learning beyond curriculum through outbound activities and creative assignments.
- Imparting contemporary and emerging techno-managerial skills to keep pace with the changing global trends.
- Facilitating greater Industry-Institute Interaction for skill development and employability enhancement.
- Establishing systems and processes to facilitate research, innovation and entrepreneurship for holistic development of students.
- Implementation of Quality Assurance System in all Institutional processes

Department of Computer Science and Engineering

Vision and Mission of the Department

Department Vision

Epitomize CSE graduate to carve a niche globally in the field of computer science to excel in the world of information technology and automation by imparting knowledge to sustain skills for the changing trends in the society and industry.

Department Mission

M1: To educate students to become excellent engineers in a confident and creative environment through world-class pedagogy.

M2: Enhancing the knowledge in the changing technology trends by giving hands-on experience through continuous education and by making them to organize & participate in various events.

M3: Impart skills in the field of IT and its related areas with a focus on developing the required competencies and virtues to meet the industry expectations.

M4: Ensure quality research and innovations to fulfill industry, government & social needs.

M5: Impart entrepreneurship and consultancy skills to students to develop self-sustaining life skills in multi-disciplinary areas.

Programme Educational Objectives

PEO 1: Engage in professional practice to promote the development of innovative systems and optimized solutions for Computer Science and Engineering.

PEO 2: Adapt to different roles and responsibilities in interdisciplinary working environment by respecting professionalism and ethical practices within organization and society at national and international level.

PEO 3: Graduates will engage in life-long learning and professional development to acclimate the rapidly changing work environment and develop entrepreneurship skills.

Program Outcomes (POs)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PSO 1: Foundation of Mathematical Concepts: Ability to use mathematical methodologies to crack problem using suitable mathematical analysis, data structure and suitable algorithm.

PSO 2: Foundation of Computer System: Ability to interpret the fundamental concepts and methodology of computer systems. Students can understand the functionality of hardware and software aspectsof computer systems.

PSO 3: Foundations of Software Development: Ability to grasp the software development lifecycle and methodologies of software systems. Possess competent skills and knowledge of software design process. Familiarity and practical proficiency with a broad area of programming concepts and provide new ideas and innovations towards research.

PSO 4: Foundations of Multi-Disciplinary Work: Ability to acquire leadership skills to perform professional activities with social responsibilities, through excellent flexibility to function in multi-disciplinary work environment with self-learning skills.

List of Programs:

Sl. No.	Experiments/Programs	COs
1	Design and implement a C program that helps a meteorological department analyse temperature data over a month. The program should perform the following tasks using arrays **Input Temperature Data:** • Create an array to store the daily temperatures (in degrees Celsius) for 30 days. • Allow the user to input the temperature for each day. **Calculate Average Temperature:** • Calculate and display the average temperature for the month. **Find Maximum and Minimum Temperatures:** • Identify and display the highest and lowest temperatures recorded during the month. • Also, display the days on which these temperatures occurred. **Days Above Average Temperature:** • Count and display the number of days on which the temperature was above the monthly average. **Expected Outcome:** The program should allow the user to input 30 temperatures, then perform the analysis and display the results in a clear and organized manner	**CO6**

Source code:

```
#include <stdio.h>
#define DAYS 30  // Constant for the number of days in a month
// Function declarations
void inputTemperatures(float temps[]);
float calculateAverage(float temps[]);
void findMaxMinTemps(float temps[], float *max, float *min, int *maxDay, int *minDay);
int countAboveAverage(float temps[], float average);
int main() {
    float temperatures[DAYS];
    float averageTemp, maxTemp, minTemp;
    int maxDay, minDay, aboveAverageCount;
    // Input temperatures for 30 days
    inputTemperatures(temperatures);
    // Calculate average temperature
    averageTemp = calculateAverage(temperatures);
    printf("\nAverage Temperature: %.2f°C\n", averageTemp);
    findMaxMinTemps(temperatures, &maxTemp, &minTemp, &maxDay, &minDay);
    printf("Maximum Temperature: %.2f°C on Day %d\n", maxTemp, maxDay + 1);
    printf("Minimum Temperature: %.2f°C on Day %d\n", minTemp, minDay + 1);

    // Count days above average temperature
    aboveAverageCount = countAboveAverage(temperatures, averageTemp);
    printf("Number of days with temperature above average: %d\n", aboveAverageCount);

    return 0;
}
void inputTemperatures(float temps[]) {
    for (int i = 0; i < DAYS; i++) {
        printf("Enter temperature for Day %d: ", i + 1);
        scanf("%f", &temps[i]);
    }
}
// Function to calculate average temperature
float calculateAverage(float temps[]) {
    float sum = 0;
    for (int i = 0; i < DAYS; i++) {
        sum += temps[i];
    }
    return sum / DAYS;
}
// Function to find the maximum and minimum temperatures and their respective days
void findMaxMinTemps(float temps[], float *max, float *min, int *maxDay, int *minDay) {
    *max = temps[0];
    *min = temps[0];
    *maxDay = 0;
    *minDay = 0;

    for (int i = 1; i < DAYS; i++) {
        if (temps[i] > *max) {
            *max = temps[i];
            *maxDay = i;
        }
```

	```
	        if (temps[i] < *min) {
	            *min = temps[i];
	            *minDay = i;
	        }
	    }
	}
	// Function to count how many days had a temperature above the average
	int countAboveAverage(float temps[], float average) {
	    int count = 0;
	    for (int i = 0; i < DAYS; i++) {
	        if (temps[i] > average) {
	            count++;
	        }
	    }
	    return count;
	}
	```

2	Design and implement a C program to help a library manage its book collection. The program should perform the following tasks using arrays:
	Store Book Information:
	• Create an array to store the titles of books available in the library.
	• Allow the user to input up to 100 book titles.
	Search for a Book:
	• Allow the user to search for a book by its title.
	• Display a message indicating whether the book is available in the library.
	Display All Books:
	• Display a list of all book titles currently stored in the library.
	Remove a Book:
	• Allow the user to remove a book from the library by entering its title.
	• Update the array accordingly and display the updated list of books.
	Expected Outcome:
	The program should allow the user to manage the library‟s book collection by adding, searching, displaying, and removing books. All operations should be performed using arrays, ensuring efficient data management.
	
	Source code:
	```
	#include <stdio.h>
	#include <string.h>
	#define MAX_BOOKS 100
	#define MAX_TITLE_LENGTH 100
	// Function declarations
	```
		CO6

```
void addBook(char books[][MAX_TITLE_LENGTH], int *count);
void searchBook(char books[][MAX_TITLE_LENGTH], int count);
void displayBooks(char books[][MAX_TITLE_LENGTH], int count);
void removeBook(char books[][MAX_TITLE_LENGTH], int *count);
int main() {
   char books[MAX_BOOKS][MAX_TITLE_LENGTH];  // Array to store up to 100 book
titles
   int bookCount = 0;                     // Number of books currently in the library
   int choice;
   do {
      // Display menu options
      printf("\nLibrary Management System:\n");
      printf("1. Add a Book\n");
      printf("2. Search for a Book\n");
      printf("3. Display All Books\n");
      printf("4. Remove a Book\n");
      printf("5. Exit\n");
      printf("Enter your choice (1-5): ");
      scanf("%d", &choice);
      getchar();  // To consume the newline character after scanf
      switch (choice) {
         case 1:
            addBook(books, &bookCount);
            break;
         case 2:
            searchBook(books, bookCount);
            break;
         case 3:
            displayBooks(books, bookCount);
            break;
         case 4:
            removeBook(books, &bookCount);
            break;
         case 5:
            printf("Exiting the program.\n");
            break;
        default:
            printf("Invalid choice. Please select a valid option.\n");
        }
    } while (choice != 5);
    return 0;
}
// Function to add a book to the library
void addBook(char books[][MAX_TITLE_LENGTH], int *count) {
    if (*count < MAX_BOOKS) {
        printf("Enter the title of the book: ");
        fgets(books[*count], MAX_TITLE_LENGTH, stdin);
        books[*count][strcspn(books[*count], "\n")] = 0;  // Remove newline character
        (*count)++;
        printf("Book added successfully.\n");
    } else {
        printf("Library is full. Cannot add more books.\n");
    }
}
// Function to search for a book in the library
void searchBook(char books[][MAX_TITLE_LENGTH], int count) {
    char title[MAX_TITLE_LENGTH];
    int found = 0;
    printf("Enter the title of the book to search: ");
    fgets(title, MAX_TITLE_LENGTH, stdin);
    title[strcspn(title, "\n")] = 0;  // Remove newline character
    for (int i = 0; i < count; i++) {
        if (strcmp(books[i], title) == 0) {
            printf("The book \"%s\" is available in the library.\n", title);
            found = 1;
            break;
        }
    }
    if (!found) {
        printf("The book \"%s\" is not available in the library.\n", title);
    }
}
// Function to display all books in the library
void displayBooks(char books[][MAX_TITLE_LENGTH], int count) {
    if (count == 0) {
        printf("No books in the library.\n");
    } else {
        printf("Books currently in the library:\n");
        for (int i = 0; i < count; i++) {
            printf("%d. %s\n", i + 1, books[i]);
        }
    }
}
// Function to remove a book from the library
void removeBook(char books[][MAX_TITLE_LENGTH], int *count) {
    char title[MAX_TITLE_LENGTH];
    int found = 0;

    if (*count == 0) {
        printf("No books to remove.\n");
        return;
    }
    printf("Enter the title of the book to remove: ");
    fgets(title, MAX_TITLE_LENGTH, stdin);
    title[strcspn(title, "\n")] = 0;  // Remove newline character

    for (int i = 0; i < *count; i++) {
        if (strcmp(books[i], title) == 0) {
            // Shift all subsequent books one position to the left
            for (int j = i; j < *count - 1; j++) {
                strcpy(books[j], books[j + 1]);
            }
            (*count)--;
            printf("The book \"%s\" has been removed from the library.\n", title);
            found = 1;
```

	```
	          break;
	        }
	    }
	    if (!found) {
	        printf("The book \"%s\" was not found in the library.\n", title);
	    }
	    // Display updated list of books
	    displayBooks(books, *count);
	}
	```
3	Design and implement a C program to simulate the navigation functionality of a web browser using stacks. The program should allow the user to: **Visit a New Website:** • Each time the user visits a new website, store the URL in a stack representing the "Back" history.	**CO6**

- Clear the "Forward" history stack when a new website is visited.

Go Back to the Previous Website:

- When the user selects "Back," pop the URL from the "Back" stack and push it onto the "Forward" stack.
- Display the URL of the previous website.

Go Forward to the Next Website:

- When the user selects "Forward," pop the URL from the "Forward" stack and push it onto the "Back" stack.
- Display the URL of the next website.

Display Current Website:

- Display the URL of the current website the user is viewing.

Expected Outcome:

The program should allow the user to navigate through websites using "Back" and "Forward" operations, just like in a real web browser. The stacks should correctly manage the history, ensuring accurate navigation.

Source code:

```
#include <stdio.h>
#include <string.h>

#define MAX_HISTORY 100
#define MAX_URL_LENGTH 100

// Stack structure for storing URLs
typedef struct {
    char urls[MAX_HISTORY][MAX_URL_LENGTH];
    int top;
} Stack;

// Function declarations
void push(Stack *stack, char *url);
char *pop(Stack *stack);
```

```
int isEmpty(Stack *stack);
void visitNewWebsite(Stack *backStack, Stack *forwardStack, char *currentWebsite);
void goBack(Stack *backStack, Stack *forwardStack, char *currentWebsite);
void goForward(Stack *backStack, Stack *forwardStack, char *currentWebsite);
void displayCurrentWebsite(char *currentWebsite);

int main() {
    Stack backStack = {.top = -1};    // Stack to store "Back" history
    Stack forwardStack = {.top = -1}; // Stack to store "Forward" history
    char currentWebsite[MAX_URL_LENGTH] = "home";  // Starting page (home)
    int choice;

    do {
        // Display menu options
        printf("\nBrowser Navigation System:\n");
        printf("1. Visit a New Website\n");
        printf("2. Go Back to Previous Website\n");
        printf("3. Go Forward to Next Website\n");
        printf("4. Display Current Website\n");
        printf("5. Exit\n");
        printf("Enter your choice (1-5): ");
        scanf("%d", &choice);
        getchar();  // To consume newline character after scanf

        switch (choice) {
            case 1: {
                char newWebsite[MAX_URL_LENGTH];
                printf("Enter the URL of the new website: ");
                fgets(newWebsite, MAX_URL_LENGTH, stdin);
                newWebsite[strcspn(newWebsite, "\n")] = 0;  // Remove newline
                visitNewWebsite(&backStack, &forwardStack, currentWebsite);
                strcpy(currentWebsite, newWebsite);
                break;
            }
            case 2:
                goBack(&backStack, &forwardStack, currentWebsite);
                break;
            case 3:
                goForward(&backStack, &forwardStack, currentWebsite);
                break;
            case 4:
                displayCurrentWebsite(currentWebsite);
                break;
            case 5:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice. Please select a valid option.\n");
        }
    } while (choice != 5);

    return 0;
}

// Function to push a URL onto a stack
void push(Stack *stack, char *url) {
    if (stack->top < MAX_HISTORY - 1) {
        stack->top++;
        strcpy(stack->urls[stack->top], url);
    } else {
        printf("Stack is full. Cannot push more URLs.\n");
    }
}

// Function to pop a URL from a stack
char *pop(Stack *stack) {
    if (!isEmpty(stack)) {
        return stack->urls[stack->top--];
    } else {
        return NULL;
    }
}

// Function to check if a stack is empty
int isEmpty(Stack *stack) {
    return stack->top == -1;
}

// Function to visit a new website
void visitNewWebsite(Stack *backStack, Stack *forwardStack, char *currentWebsite) {
    // Push the current website onto the back stack
    push(backStack, currentWebsite);
    // Clear the forward stack
    forwardStack->top = -1;
    printf("Visited a new website.\n");
}

// Function to go back to the previous website
void goBack(Stack *backStack, Stack *forwardStack, char *currentWebsite) {
    if (!isEmpty(backStack)) {
        // Push the current website onto the forward stack
        push(forwardStack, currentWebsite);
        // Pop the previous website from the back stack and update currentWebsite
        strcpy(currentWebsite, pop(backStack));
        printf("Went back to: %s\n", currentWebsite);
    } else {
        printf("No more history to go back.\n");
    }
}
// Function to go forward to the next website
void goForward(Stack *backStack, Stack *forwardStack, char *currentWebsite) {
    if (!isEmpty(forwardStack)) {
        // Push the current website onto the back stack
        push(backStack, currentWebsite);
        // Pop the next website from the forward stack and update currentWebsite
        strcpy(currentWebsite, pop(forwardStack));
        printf("Went forward to: %s\n", currentWebsite);
    } else {
        printf("No more forward history.\n");
    }
}
// Function to display the current website
void displayCurrentWebsite(char *currentWebsite) {
    printf("Current website: %s\n", currentWebsite);
}
```

4 Design, Develop and Implement a Program in C for the following Stack Applications **CO6**

a. Evaluation of Suffix expression with single digit operands and operators: +, -, *, /, %, ^

Solving Tower of Hanoi problem with n disks

a) Source code:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int i, top = -1;
int op1, op2, res, s[20];
char postfix[90], symb;
void push(int item)
{
        top = top+1;
        s[top] = item;
}
int pop()
{
        int item;
        item  =  s[top];
        top = top-1;
        return item;
}
void main()
{
        printf("\nEnter a valid postfix expression:\n");
        scanf("%s", postfix);
        for(i=0; postfix[i]!='\0'; i++)
        {
                symb = postfix[i];
                if(isdigit(symb))
                {
                        push(symb - '0');
                }
                else
                {
                        op2 = pop();
                        op1 = pop();
                        switch(symb)
                        {
                                case '+':       push(op1+op2);
                                                break;
                                case '-':       push(op1-op2);
                                                break;
                                case '*':       push(op1*op2);
                                                break;
                                case '/':       push(op1/op2);
                                                break;
                                case '%':       push(op1%op2);
                                                break;
                                case '$':
                                case '^':       push(pow(op1, op2));
                                                break;
                                default :   push(0);
                        }
                }
        }
        res = pop();
        printf("\n Result = %d", res);
}
```

b)Source code:

```
#include <stdio.h>
// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
        if (n == 1)
        {
                printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
                return;
        }
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
        int n = 4; // Number of disks
        towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
        return 0;
}
```

5

Design and implement a C program to manage a list of patients in a hospital usinga singly linked list. The program should support the following functionalities:

CO6

Add a New Patient:

- Each patient has a unique ID, name, and age.
- Allow the user to add a new patient at the end of the list.

Display All Patients:

- Display the details (ID, name, age) of all patients currently in the list.

Search for a Patient by ID:

- Allow the user to search for a patient by their unique ID and display their details if found.

Delete a Patient Record:

- Allow the user to delete a patient record by their ID.
- Ensure that the linked list is updated correctly after deletion.

Count Total Number of Patients:

- Display the total number of patients currently in the system.

Expected Outcome:

The program should allow the user to effectively manage a list of patients, adding new entries, displaying all records, searching for specific patients, deleting records, and counting the total number of patients.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Define the structure for a Patient
struct Patient {
    int id;
    char name[50];
    int age;
    struct Patient* next;
};

// Initialize head pointer to NULL
struct Patient* head = NULL;
// Function to add a new patient at the end of the list
void addPatient(int id, char name[], int age) {
    struct Patient* newPatient = (struct Patient*)malloc(sizeof(struct Patient));
    newPatient->id = id;
    strcpy(newPatient->name, name);
    newPatient->age = age;
    newPatient->next = NULL;
    if (head == NULL) {
        head = newPatient;  // If list is empty, new patient becomes the head
    } else {
        struct Patient* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;  // Traverse to the last node
        }
        temp->next = newPatient;  // Link the new patient at the end
    }
}
// Function to display all patients in the list
void displayPatients() {
    struct Patient* temp = head;
    if (temp == NULL) {
        printf("No patients in the list.\n");
        return;
    }
    printf("Patient List:\n");
    while (temp != NULL) {
        printf("ID: %d, Name: %s, Age: %d\n", temp->id, temp->name, temp->age);
        temp = temp->next;
    }
}
// Function to search for a patient by ID
void searchPatient(int id) {
    struct Patient* temp = head;
    while (temp != NULL) {
        if (temp->id == id) {
            printf("Patient found - ID: %d, Name: %s, Age: %d\n", temp->id, temp->name, temp-
>age);
            return;
        }
        temp = temp->next;
    }
    printf("Patient with ID %d not found.\n", id);
```

```
}
// Function to delete a patient by ID
void deletePatient(int id) {
    struct Patient* temp = head;
    struct Patient* prev = NULL;
    if (temp != NULL && temp->id == id) {
        head = temp->next;  // Patient to be deleted is the head
        free(temp);
        printf("Patient with ID %d deleted.\n", id);
        return;
    }
    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Patient with ID %d not found.\n", id);
        return;
    }
    prev->next = temp->next;  // Unlink the patient from the list
    free(temp);
    printf("Patient with ID %d deleted.\n", id);
}
// Function to count the total number of patients
void countPatients() {
    struct Patient* temp = head;
    int count = 0;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    printf("Total number of patients: %d\n", count);
}

int main() {
```

```
int choice, id, age;
char name[50];

while (1) {
    printf("\nHospital Patient Management System\n");
    printf("1. Add New Patient\n");
    printf("2. Display All Patients\n");
    printf("3. Search for Patient by ID\n");
    printf("4. Delete Patient Record\n");
    printf("5. Count Total Patients\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter Patient ID: ");
            scanf("%d", &id);
            printf("Enter Patient Name: ");
            scanf("%s", name);
            printf("Enter Patient Age: ");
            scanf("%d", &age);
            addPatient(id, name, age);
            break;
        case 2:
            displayPatients();
            break;
        case 3:
            printf("Enter Patient ID to search: ");
            scanf("%d", &id);
            searchPatient(id);
            break;
        case 4:
            printf("Enter Patient ID to delete: ");
            scanf("%d", &id);
            deletePatient(id);
            break;
          case 5:
            countPatients();
            break;
          case 6:
            printf("Exiting the program.\n");
            exit(0);
          default:
            printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

6	Design a C program to perform the addition of two polynomials (2 variable polynomial) using a doubly linked list. The program should be able to handle polynomials with both positive and negative coefficients.	**CO6**

Source code:

```
#include <stdio.h>
#include <stdlib.h>

// Structure for a polynomial term
struct Term {
    int coeff;   // Coefficient
    int exp_x;   // Exponent of x
    int exp_y;   // Exponent of y
    struct Term* next;
    struct Term* prev;
};

// Function to create a new polynomial term
struct Term* createTerm(int coeff, int exp_x, int exp_y) {
    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
    newTerm->coeff = coeff;
    newTerm->exp_x = exp_x;
    newTerm->exp_y = exp_y;
    newTerm->next = NULL;
    newTerm->prev = NULL;
    return newTerm;
}

// Function to add a term to the end of the polynomial list
void addTerm(struct Term** head, int coeff, int exp_x, int exp_y) {
    struct Term* newTerm = createTerm(coeff, exp_x, exp_y);

    if (*head == NULL) {
        *head = newTerm;  // If the list is empty, new term becomes the head
    } else {
        struct Term* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;  // Traverse to the end of the list
        }
        temp->next = newTerm;
        newTerm->prev = temp;
    }
}

// Function to display the polynomial
void displayPolynomial(struct Term* head) {
    if (head == NULL) {
        printf("0\n");
        return;
    }
    struct Term* temp = head;
    while (temp != NULL) {
        if (temp->coeff > 0 && temp != head) {
            printf(" + ");
        } else if (temp->coeff < 0) {
            printf(" - ");
        }
        printf("%d", abs(temp->coeff));
        if (temp->exp_x != 0) {
            printf("x^%d", temp->exp_x);
        }
        if (temp->exp_y != 0) {
            printf("y^%d", temp->exp_y);
        }
        temp = temp->next;
    }
    printf("\n");
}

// Function to add two polynomials
struct Term* addPolynomials(struct Term* p1, struct Term* p2) {
    struct Term* result = NULL;

    while (p1 != NULL && p2 != NULL) {
        if (p1->exp_x == p2->exp_x && p1->exp_y == p2->exp_y) {
            // If both terms have the same exponents, add the coefficients
            int sumCoeff = p1->coeff + p2->coeff;
            if (sumCoeff != 0) {
                addTerm(&result, sumCoeff, p1->exp_x, p1->exp_y);
            }
            p1 = p1->next;
            p2 = p2->next;
        } else if (p1->exp_x > p2->exp_x || (p1->exp_x == p2->exp_x && p1->exp_y > p2-
>exp_y)) {
            // If p1 term has a higher exponent, add it to the result
            addTerm(&result, p1->coeff, p1->exp_x, p1->exp_y);
            p1 = p1->next;
        } else {
            // If p2 term has a higher exponent, add it to the result
            addTerm(&result, p2->coeff, p2->exp_x, p2->exp_y);
            p2 = p2->next;
        }
    }

    // Add remaining terms of p1, if any
    while (p1 != NULL) {
        addTerm(&result, p1->coeff, p1->exp_x, p1->exp_y);
        p1 = p1->next;
    }

    // Add remaining terms of p2, if any
    while (p2 != NULL) {
        addTerm(&result, p2->coeff, p2->exp_x, p2->exp_y);
        p2 = p2->next;
    }

    return result;
}

// Function to take user input for a polynomial
void inputPolynomial(struct Term** head) {
    int n;
    printf("Enter the number of terms in the polynomial: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        int coeff, exp_x, exp_y;
        printf("Enter coefficient, exponent of x, and exponent of y for term %d: ", i+1);
        scanf("%d %d %d", &coeff, &exp_x, &exp_y);
        addTerm(head, coeff, exp_x, exp_y);
    }
}

int main() {
    struct Term* poly1 = NULL;
    struct Term* poly2 = NULL;
    // Take input for the first polynomial
    printf("Input first polynomial:\n");
    inputPolynomial(&poly1);
       // Take input for the second polynomial
    printf("Input second polynomial:\n");
    inputPolynomial(&poly2);
    // Display the polynomials
    printf("First Polynomial: \n");
    displayPolynomial(poly1);
    printf("Second Polynomial: \n");
    displayPolynomial(poly2);
    // Add the two polynomials
    struct Term* result = addPolynomials(poly1, poly2);
    printf("Resultant Polynomial after Addition: \n");
    displayPolynomial(result);
    return 0;
}
```

7	Given an array of elements, construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0. Ex: Input : arr[] = {1, 2, 3, 4, 5, 6}	**CO6**

Source code:

```
. #include <stdio.h>
#include <stdlib.h>

// Structure of a tree node
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function to create a new tree node
struct TreeNode* createNode(int data) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert nodes into the binary tree level by level
struct TreeNode* insertLevelOrder(int arr[], struct TreeNode* root, int i, int n) {
    // Base case for recursion
    if (i < n) {
        struct TreeNode* temp = createNode(arr[i]);
        root = temp;

        // Insert left child
        root->left = insertLevelOrder(arr, root->left, 2 * i + 1, n);

        // Insert right child
        root->right = insertLevelOrder(arr, root->right, 2 * i + 2, n);
    }
    return root;
}

// Function to print the tree in level order (using in-order traversal for testing purposes)
void inOrderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

// Function to perform level-order traversal of the tree using a queue
void printLevelOrder(struct TreeNode* root) {
    if (root == NULL) return;

    // Create a queue
    struct TreeNode** queue = (struct TreeNode**)malloc(100 * sizeof(struct TreeNode*));
    int front = 0, rear = 0;

    // Enqueue root
    queue[rear++] = root;

    while (front < rear) {
        struct TreeNode* node = queue[front++];

        // Print current node
        printf("%d ", node->data);

        // Enqueue left child
        if (node->left != NULL)
            queue[rear++] = node->left;

        // Enqueue right child
        if (node->right != NULL)
            queue[rear++] = node->right;
    }
```

	```
	    free(queue);
	}

	int main() {
	    // Example input array
	    int arr[] = {1, 2, 3, 4, 5, 6, 7};
	    int n = sizeof(arr) / sizeof(arr[0]);

	    // Create the binary tree from the array
	    struct TreeNode* root = insertLevelOrder(arr, root, 0, n);

	    // Print the tree in level order to verify the structure
	    printf("Level order traversal of the constructed binary tree: \n");
	    printLevelOrder(root);

	    return 0;
	}
	```
8	Write a C program to manage student records using a Binary Search Tree (BST). Each student record should include a unique roll number, name, and GPA. The	**CO6**

BST should be organized by the roll number, which will be used as the key.

Operations to Implement:
Insert Record: Insert a new student record into the BST. Ensure that roll numbers are unique.
Search Record: Search for a student record by roll number. If found, display the student's name and GPA.
Delete Record: Delete a student record by roll number. Ensure that the BST properties are maintained after deletion.
In-order Traversal: Display all student records in ascending order of roll numbers (in-order traversal).
Display Students with GPA Above a Certain Threshold: Implement a function to display all students whose GPA is above a user-defined threshold.

Source code:
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to represent a student record
struct Student {
    int rollNo;
    char name[100];
    float gpa;
    struct Student* left;
    struct Student* right;
};

// Function to create a new student node
struct Student* createStudent(int rollNo, char name[], float gpa) {
    struct Student* newStudent = (struct Student*)malloc(sizeof(struct Student));
    newStudent->rollNo = rollNo;
    strcpy(newStudent->name, name);
    newStudent->gpa = gpa;
    newStudent->left = NULL;
    newStudent->right = NULL;
    return newStudent;
}

// Insert a student record into the BST
struct Student* insertStudent(struct Student* root, int rollNo, char name[], float gpa) {
    // If the tree is empty, create a new student node and return it
    if (root == NULL) {
        return createStudent(rollNo, name, gpa);
    }

    // If the roll number is smaller, insert in the left subtree
    if (rollNo < root->rollNo) {
        root->left = insertStudent(root->left, rollNo, name, gpa);
    }
    // If the roll number is greater, insert in the right subtree
    else if (rollNo > root->rollNo) {
        root->right = insertStudent(root->right, rollNo, name, gpa);
    }
    // If the roll number already exists, do nothing (unique roll number required)
    else {
        printf("Student with roll number %d already exists!\n", rollNo);
    }

    return root;
}

// Function to search for a student by roll number
struct Student* searchStudent(struct Student* root, int rollNo) {
    // Base case: root is null or rollNo is present at root
    if (root == NULL || root->rollNo == rollNo) {
        return root;
    }

    // Roll number is greater, search in the right subtree
    if (rollNo > root->rollNo) {
        return searchStudent(root->right, rollNo);
    }

    // Roll number is smaller, search in the left subtree
    return searchStudent(root->left, rollNo);
}

// Function to perform in-order traversal (display students in ascending order of rollNo)
void inOrderTraversal(struct Student* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("Roll No: %d, Name: %s, GPA: %.2f\n", root->rollNo, root->name, root->gpa);
        inOrderTraversal(root->right);
    }
}

// Function to find the minimum value node in a BST
struct Student* findMinValueNode(struct Student* node) {
    struct Student* current = node;
    // Loop down to find the leftmost leaf (minimum value)
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}

// Function to delete a student by roll number
struct Student* deleteStudent(struct Student* root, int rollNo) {
    if (root == NULL) return root;

    // If the rollNo to be deleted is smaller, go to the left subtree
    if (rollNo < root->rollNo) {
        root->left = deleteStudent(root->left, rollNo);
    }
    // If the rollNo to be deleted is greater, go to the right subtree
    else if (rollNo > root->rollNo) {
        root->right = deleteStudent(root->right, rollNo);
    }
    // If rollNo matches, this is the node to be deleted
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Student* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Student* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: get the inorder successor (smallest in the right subtree)
        struct Student* temp = findMinValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->rollNo = temp->rollNo;
        strcpy(root->name, temp->name);
        root->gpa = temp->gpa;

        // Delete the inorder successor
        root->right = deleteStudent(root->right, temp->rollNo);
    }

    return root;
}

// Function to display students with GPA above a certain threshold
void displayStudentsAboveGPA(struct Student* root, float threshold) {
    if (root != NULL) {
        displayStudentsAboveGPA(root->left, threshold);
        if (root->gpa > threshold) {
            printf("Roll No: %d, Name: %s, GPA: %.2f\n", root->rollNo, root->name, root->gpa);
        }
        displayStudentsAboveGPA(root->right, threshold);
    }
}

int main() {
    struct Student* root = NULL;
    int choice, rollNo;
    char name[100];
    float gpa, threshold;

    while (1) {
        printf("\nStudent Record Management System\n");
        printf("1. Insert Student Record\n");
        printf("2. Search Student Record\n");
        printf("3. Delete Student Record\n");
        printf("4. Display All Student Records (In-order Traversal)\n");
        printf("5. Display Students with GPA Above Threshold\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Roll No: ");
                scanf("%d", &rollNo);
                printf("Enter Name: ");
                scanf("%s", name);
                printf("Enter GPA: ");
                scanf("%f", &gpa);
                root = insertStudent(root, rollNo, name, gpa);
                break;

            case 2:
                printf("Enter Roll No to Search: ");
                scanf("%d", &rollNo);
                struct Student* student = searchStudent(root, rollNo);
                if (student != NULL) {
                    printf("Student Found - Roll No: %d, Name: %s, GPA: %.2f\n", student->rollNo,
student->name, student->gpa);
                } else {
                    printf("Student with Roll No %d not found!\n", rollNo);
                }
                break;

            case 3:
                printf("Enter Roll No to Delete: ");
                scanf("%d", &rollNo);
                root = deleteStudent(root, rollNo);
                break;

            case 4:
                printf("Displaying All Student Records (In-order Traversal):\n");
                inOrderTraversal(root);
                break;

            case 5:
                printf("Enter GPA Threshold: ");
                scanf("%f", &threshold);
                printf("Students with GPA above %.2f:\n", threshold);
                displayStudentsAboveGPA(root, threshold);
                break;

            case 6:
                exit(0);

            default:
                printf("Invalid Choice! Please try again.\n");
        }
    }

    return 0;
}
```

9 Write a C program to manage course prerequisites in a university using a Directed Acyclic Graph (DAG). Each course is represented as a node, and a directed edge from course A to course B indicates that course A is a prerequisite for course B. **CO6**

Operations to Implement:

Add Course: Add a new course (node) to the graph.

Add Prerequisite: Create a prerequisite relationship (directed edge) between two courses.

Remove Prerequisite: Remove an existing prerequisite relationship between two courses.

Display Courses and Their Prerequisites: Display all courses and their direct prerequisites.

Topological Sorting: Implement a function to determine the order in whichcourses should be taken based on their prerequisites using topological sorting.

Check for Cycles: Implement a function to check if the graph has any cycles (i.e., a circular dependency between courses) since the graph must remain acyclic

Source code:

```
 #include <stdio.h>
#include <stdlib.h>

#define MAX_COURSES 100

// Graph structure
struct Graph {
   int numCourses;
   int adjMatrix[MAX_COURSES][MAX_COURSES]; // Adjacency matrix to represent the
graph
};

// Function to create a new graph with a specified number of courses
struct Graph* createGraph(int numCourses) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numCourses = numCourses;

   // Initialize the adjacency matrix with 0s (no edges)
   for (int i = 0; i < numCourses; i++) {
      for (int j = 0; j < numCourses; j++) {
         graph->adjMatrix[i][j] = 0;
      }
   }

   return graph;
}

// Add a prerequisite (directed edge from courseA to courseB)
void addPrerequisite(struct Graph* graph, int courseA, int courseB) {
   graph->adjMatrix[courseA][courseB] = 1;
}

// Remove a prerequisite (remove the directed edge from courseA to courseB)
void removePrerequisite(struct Graph* graph, int courseA, int courseB) {
   graph->adjMatrix[courseA][courseB] = 0;
}

// Display all courses and their prerequisites
void displayPrerequisites(struct Graph* graph) {
    for (int i = 0; i < graph->numCourses; i++) {
        printf("Course %d prerequisites: ", i);
        for (int j = 0; j < graph->numCourses; j++) {
            if (graph->adjMatrix[i][j] == 1) {
                printf("%d ", j);
            }
        }
        printf("\n");
    }
}

// Utility function for topological sorting
void topologicalSortUtil(struct Graph* graph, int course, int visited[], int* stack, int*
stackIndex) {
    visited[course] = 1; // Mark the current node as visited

    // Recur for all the vertices adjacent to this vertex
    for (int i = 0; i < graph->numCourses; i++) {
        if (graph->adjMatrix[course][i] == 1 && !visited[i]) {
            topologicalSortUtil(graph, i, visited, stack, stackIndex);
        }
    }

    // Push the current course to the stack (completed)
    stack[(*stackIndex)--] = course;
}

// Function to perform topological sorting
void topologicalSort(struct Graph* graph) {
    int visited[MAX_COURSES] = {0};
    int stack[MAX_COURSES];
    int stackIndex = graph->numCourses - 1;

    // Perform DFS for all courses to find topological order
    for (int i = 0; i < graph->numCourses; i++) {
        if (!visited[i]) {
            topologicalSortUtil(graph, i, visited, stack, &stackIndex);
        }
    }

    // Print the courses in topological order
    printf("Topological Sort (Course Order):\n");
    for (int i = 0; i < graph->numCourses; i++) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

// Utility function for cycle detection using DFS
int isCyclicUtil(struct Graph* graph, int course, int visited[], int recStack[]) {
    if (!visited[course]) {
        visited[course] = 1;
        recStack[course] = 1;

        // Recur for all vertices adjacent to this vertex
        for (int i = 0; i < graph->numCourses; i++) {
            if (graph->adjMatrix[course][i]) {
                if (!visited[i] && isCyclicUtil(graph, i, visited, recStack)) {
                    return 1;
                } else if (recStack[i]) {
                    return 1;
                }
            }
        }
    }

    recStack[course] = 0; // Remove the course from recursion stack
    return 0;
}

// Function to detect a cycle in the graph
int isCyclic(struct Graph* graph) {
    int visited[MAX_COURSES] = {0};
    int recStack[MAX_COURSES] = {0};

    // Check for cycle starting from each course
    for (int i = 0; i < graph->numCourses; i++) {
        if (isCyclicUtil(graph, i, visited, recStack)) {
            return 1; // Graph contains cycle
        }
    }
    return 0; // No cycle
}

int main() {
    int numCourses, choice, courseA, courseB;
    printf("Enter the number of courses: ");
    scanf("%d", &numCourses);

    // Create the graph
    struct Graph* graph = createGraph(numCourses);

    while (1) {
        printf("\nUniversity Course Management\n");
        printf("1. Add Prerequisite (Edge)\n");
        printf("2. Remove Prerequisite (Edge)\n");
        printf("3. Display Courses and Their Prerequisites\n");
        printf("4. Topological Sort (Course Order)\n");
        printf("5. Check for Cycles (Circular Dependency)\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Course A (prerequisite) and Course B: ");
                scanf("%d %d", &courseA, &courseB);
                addPrerequisite(graph, courseA, courseB);
                break;

            case 2:
                printf("Enter Course A (prerequisite) and Course B to remove: ");
                scanf("%d %d", &courseA, &courseB);
                removePrerequisite(graph, courseA, courseB);
                break;

            case 3:
                displayPrerequisites(graph);
                break;

            case 4:
                topologicalSort(graph);
                break;

            case 5:
                if (isCyclic(graph)) {
                    printf("The graph contains a cycle (circular dependency)!\n");
                } else {
                    printf("No cycles (circular dependency) detected.\n");
                }
                break;

            case 6:
                exit(0);

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

    return 0;
}
```

10 Design and develop a program in C that uses Hash Function H:K->L as H(K)=K mod m(reminder method) and implement hashing technique to map a given key Kto the address space L. Resolve the collision (if any) using linear probing. **CO6**

Source code:

```
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10  // Size of the hash table

// Function to implement the hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Function to insert a key into the hash table using linear probing
void insert(int hashTable[], int key) {
    int index = hashFunction(key);

    // Linear probing in case of collision
    while (hashTable[index] != -1) {
        index = (index + 1) % TABLE_SIZE;  // Move to the next index
    }

    hashTable[index] = key;  // Insert the key at the found position
}

// Function to display the hash table
void display(int hashTable[]) {
    printf("Hash Table:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != -1)
            printf("Index %d: %d\n", i, hashTable[i]);
        else
            printf("Index %d: Empty\n", i);
    }
}

int main() {
    int hashTable[TABLE_SIZE];

    // Initialize the hash table to -1 (indicating empty slots)
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = -1;
    }
    int numKeys;
    // Ask the user for the number of keys they want to insert
    printf("Enter the number of keys to insert: ");
    scanf("%d", &numKeys);
    // Dynamically create an array to store user input
    int keys[numKeys];
    // Get the keys from the user
    printf("Enter the keys:\n");
    for (int i = 0; i < numKeys; i++) {
        printf("Key %d: ", i + 1);
        scanf("%d", &keys[i]);
    }
    // Insert the keys into the hash table
    for (int i = 0; i < numKeys; i++) {
        insert(hashTable, keys[i]);
    }
    // Display the hash table
    display(hashTable);

    return 0;
}
```

	Open ended Programs	
1	Develop a project to demonstrate the usage of any Data structures	**CO6**