



**ATME College of Engineering,
13th KM, Mysore – Kanakapura – Bangalore Road, Mysore – 570 028,**

**Department of CSE–Data Science
(Academic Year 2026-27)**



LABORATORY MANUAL

SUBJECT: DATA STRUCTURES LAB

SUB CODE: 1BCSL306

SEMESTER: III

SCHEME: 2025

Prepared By

**Dr Neethi M V
Assistant Professor**

Approved by

**Dr. Anitha D B
HoD CSE-DSE**

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING AND ENGINEERING
(DATA SCIENCE)

Vision of The Department

- To impart technical education in the field of data science of excellent quality with a high level of professional competence, social responsibility, and global awareness among the students

Mission

- To impart technical education that is up to date, relevant and makes students competitive and employable at global level
- To provide technical education with a high sense of discipline, social relevance in an intellectually, ethically and socially challenging environment for better tomorrow
- Educate to the global standards with a benchmark of excellence and to kindle the spirit of innovation.

Program Outcomes(PO)

- **PO1:** Engineering Knowledge
- **PO2:** Problem Analysis
- **PO3:** Design/Development of Solutions
- **PO4:** Conduct Investigations of Complex Problems
- **PO5:** Engineering Tool Usage (renamed from Modern Tool Usage)
- **PO6:** The Engineer and the World (merged old PO6 Engineer and Society with old PO7 Environment and Sustainability)
- **PO7:** Ethics (expanded as a standalone focus)
- **PO8:** Individual and Collaborative Teamwork
- **PO9:** Communication
- **PO10:** Project Management and Finance
- **PO11:** Life-Long Learning

Program Specific Outcomes (PSOs)

- PSO1: Develop relevant programming skills to become a successful data scientist
- PSO2: Apply data science concepts and algorithms to solve real world problems of the society
- PSO3: Apply data science techniques in the various domains like agriculture, education healthcare for better society

Program Educational Objectives (PEOs):

PEO1: Develop cutting-edge skills in data science and its related technologies, such as machine learning, predictive analytic, and data engineering.

PEO2: Design and develop data-driven solutions to real-world problems in a business, research, or social environment.

PEO3: Apply data engineering and data visualization techniques to discover, investigate, and interpret data.

PEO4: Demonstrate ethical and responsible data practices in problem solving

PEO5: Integrate fields within computer science, optimization, and statistics to develop better solutions

DATA STRUCTURES LABORATORY		Scheme	2025
Course Code	1BCSL306	Semester	3
Teaching Hours/Week (L:T:P)	0:0:2	CIE Marks	50
Total Hours of Pedagogy L:T:P:SL:TW	0:0:28:2:30	SEE Marks	50
Credits	1	Total Marks	100
Examination type (SEE)	Practical	Exam Hours	03
Course outcome (Course Skill Set) At the end of the course, the student will be able to: 1. Develop C programs to implement fundamental data structures and their operations for solving computational problems. 2. Make use of appropriate data structures to organize, store, and manipulate data for solving real-world problems using C. 3. Build a laboratory record consisting of problem statements, algorithm/design, implementation, program outputs, and result with appropriate inferences.			
Note: 1. The laboratory syllabus consists of PART-A and PART-B. While PART-A has 6 conventional experiments, PART-B has 6 typical open-ended experiments. The maximum marks for the laboratory course are 100. 2. Both PART-A and PART-B are considered for CIE and SEE. 3. Students have answer 1(one) question from PART-A and 1(one) question from PART-B. a. The questions set for SEE shall be from among the experiments under PART-A. It is evaluated for 70 marks out of the maximum 100 marks. b. The open-ended question set for SEE shall be any other open-ended question and not selected from the experiments under PART-A. It shall be evaluated for 30 marks. 4. For Continuous Internal Evaluation (CIE) during the semester, classwork shall typically include open-ended questions from Part-B, along with other similar questions designed to enhance the students' skills.			
PART – A FIXED SET OF EXPERIMENTS			
1. Develop a Program in C for the following: a. Define a structure Book with the following fields: Book_ID, Title, Author, Price, Availability Status (Available/Issued) b. Dynamically allocate memory to store the details of N books, where N is entered by the user. c. Implement the following functions: i. create() – To allocate memory and read the details of N books and store them in the allocated memory. ii. display() – To display the details of all books available in the library. iii. search() – To search for a book using Book ID and display the book details if found. iv. issueBook() – To issue a book by updating its availability status and display an appropriate message. v. returnBook() – To return an issued book by updating its availability status and display an appropriate message. d. Provide a menu-driven interface with the following options: i. Add Book Records ii. Display All Book Records			

- iii. Search Book by Book ID
- iv. Issue a Book
- v. Return a Book
- vi. Exit

2. Develop a menu driven Program in C for the following operations on STACK of Integers (Array Implementation of Stack with maximum size MAX)

- a. Push an Element on to Stack
- b. Pop an Element from Stack
- c. Demonstrate how Stack can be used to check Palindrome
- d. Demonstrate Overflow and Underflow situations on Stack
- e. Display the status of Stack
- f. Exit

Support the program with appropriate functions for each of the above operations

3. Develop a program in C for simulating a Printer Queue using Queue data structure with the following operations:

- a. Add a print job to the Printer Queue.
- b. Process (delete) a print job from the Printer Queue.
- c. Display all pending print jobs in the Queue.
- d. Display the number of print jobs currently waiting in the Queue.
- e. Demonstrate Queue overflow and underflow conditions.
- f. Exit.

4. Develop a menu-driven Program in C for the following operations on a Singly Linked List of Student Records:

- a. Create a linked list using front insertion.
- b. Display all student records and count the number of nodes.
- c. Insert and delete nodes at the beginning and end of the list.
- d. Search for a student record using USN.

5. Develop a Program in C for the following:

- a. Create a Binary Tree by inserting integer elements entered by the user in level order.
- b. Implement the following functions:
 - i. createTree() – Create a binary tree using dynamic memory allocation.
 - ii. preorder() – Display the elements using Preorder Traversal.
 - iii. inorder() – Display the elements using Inorder Traversal.
 - iv. postorder() – Display the elements using Postorder Traversal.
- c. Provide a menu-driven interface with the following options:

- i. Create Binary Tree
- ii. Display Preorder Traversal
- iii. Display Inorder Traversal
- iv. Display Postorder Traversal
- v. Exit

6. Develop a Program in C for the following operations on Graph(G) of Cities

- a. Create an Adjacency Matrix representing the graph of N cities
- b. Print all the nodes reachable from a given starting node in a digraph using DFS/BFS method.

PART – B

OPEN ENDED EXPERIMENTS

Open-ended experiments are a type of laboratory activity where the outcome is not predetermined and students are given the freedom to explore, design, and conduct the experiment based on the problem statements as per the concepts defined by the course coordinator. It encourages creativity, critical thinking, and inquiry-based learning.

1. Develop a C program to:

- a. Read two sparse matrices of the same order.
- b. Convert both into 3-tuple representation.
- c. Perform matrix addition.
- d. Display the resultant matrix in both normal matrix form and 3-tuple form.

2. Develop a C-based Expression Conversion tool that uses stack data structure to convert arithmetic expressions from infix notation to postfix notation. The tool should support alphanumeric operands, parentheses, and the operators (+, -, *, /, %, ^). Analyze the correctness of the conversion by testing various valid expressions.

3. Develop a menu-driven Program in C for the following operations on Circular Queue (integer/floating point data):

- a. Insert an element into the Circular Queue.
- b. Delete an element from the Circular Queue.
- c. Display the contents of the Circular Queue.
- d. Demonstrate overflow and underflow conditions.
- e. Exit.

4. Develop a menu driven Program in C for the following operations on Doubly Linked List (DLL) of Employee Data with the fields: SSN, Name, Dept, Designation, Sal, PhNo etc.

- a. Create a DLL of N Employees Data by using end insertion.
- b. Display the status of DLL and count the number of nodes in it
- c. Perform Insertion and Deletion at End of DLL
- d. Perform Insertion and Deletion at Front of DLL

- e. Demonstrate how this DLL can be used as Double Ended Queue.
- f. Exit

5. Develop a menu driven Program in C for the following operations on Binary Search Tree (BST) of Integers.

- a. Create a BST of N Integers
- b. Traverse the BST in Inorder, Preorder and Post Order
- c. Search the BST for a given element (KEY) and report the appropriate message
- d. Exit

6. Develop a Program in C for an Employee Record Management System using Hashing with the following operations:

- a. Create a hash table to store employee records using Employee ID as the key.
- b. Insert employee records into the hash table using the Hash function $H: K \rightarrow L$ as $H(K) = K \bmod m$
- c. Resolve collisions using Linear Probing.
- d. Search for an employee record using a given Employee ID and display the details.
- e. Display the contents of the hash table along with the corresponding memory locations.

Suggested Learning Resources: (Text Book/ Reference Book/ Manuals):

Text books:

- 1. Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2023.

Reference books / Manuals:

- 1. Reema Thareja, Data Structures using C, 3rd Ed, Oxford press, 2023.
- 2. A M Tenenbaum, Data Structures using C, PHI, 1989
- 3. Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1st Ed, McGraw Hill, 2014.

Web links and Video Lectures (e-Resources):

- 1. GNU-GCC Documentation: <https://gcc.gnu.org/>
- 2. Virtual Labs on Data Structures-1: <https://ds1-iiith.vlabs.ac.in/List%20of%20experiments.html>
- 3. DSA Tutorial: <https://www.geeksforgeeks.org/dsa/dsa-tutorial-learn-data-structures-and-algorithms/>
- 4. NPTEL Course on DSA: <https://nptel.ac.in/courses/106106130>
- 5. NPTEL Course on Introduction to C Programming: <https://nptel.ac.in/courses/106104128>

Teaching-Learning Process (Innovative Delivery Methods):

The following are sample strategies that educators may adopt to enhance the effectiveness of the teaching-learning process and facilitate the achievement of course outcomes.

- 1. Engineering tool usage for the conduction of experiment
- 2. Demonstration through ICT tools
- 3. Use of virtual labs

CONTENTS

[illegible]

Introduction to Data Structure

Basic Concepts

The logical or mathematical model of a particular organization of data is called data structures. Data structures is the study of logical relationship existing between individual data elements, the way the data is organized in the memory and the efficient way of storing, accessing and manipulating the data elements.

Data Structures can be classified as:

- Primitive data structures
- Non-Primitive data structures.

Primitive data structures are the basic data structures that can be directly manipulated/operated by machine instructions. Some of these are character, integer, real, pointers etc.

Non-primitive data structures are derived from primitive data structures, they cannot be directly manipulated/operated by machine instructions, and these are group of homogeneous or heterogeneous data items. Some of these are Arrays, stacks, queues, trees, graphs etc.

Data structures are also classified as

- Linear data structures
- Non-Linear data structures.

In the Linear data structures processing of data items is possible in linear fashion, i.e., data can be processed one by one sequentially.

Example of such data structures are:

- Array
- Linked list
- Stacks
- Queues

A data structure in which insertion and deletion is not possible in a linear fashion is called as non linear data structure. i.e., which does not show the relationship of logical adjacency between the elements is called as non-linear data structure. Such as trees, graphs and files.

Data structure operations:

The particular data structures that one chooses for a given situation depends largely on the frequency with which specific operations are performed.

The following operations play major role in the processing of data.

- i) Traversing.
- ii) Searching.
- iii) Inserting.
- iv) Deleting.
- v) Sorting.
- vi) Merging

STACKS:

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at the same end, called the TOP of the stack. A stack is a non-primitive linear data structure. 1 2 3 4 5

As all the insertion and deletion are done from the same end, the first element inserted into the stack is the last element deleted from the stack and the last element inserted into the stack is the first element to be deleted. Therefore, the stack is called Last-In First-Out (LIFO) data structure.

QUEUES:

A queue is a non-primitive linear data structure. Where the operation on the queue is based on First-In-First-Out FIFO process — the first element in the queue will be the first one out. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be removed.

For inserting elements into the queue are done from the rear end and deletion is done from the front end, we use external pointers called as rear and front to keep track of the status of the queue. During insertion, Queue Overflow condition has to be checked. Likewise during deletion, Queue Underflow condition is checked.

APPLICATION OF QUEUE

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

LINKED LIST

Disadvantages of static/sequential allocation technique:

- If an item has to be deleted then all the following items will have to be moved by one allocation. Wastage of time.
- Inefficient memory utilization.
- If no consecutive memory (free) is available, execution is not possible.

Linear Linked Lists

Types of Linked lists:

- Single Linked lists
- Circular Single Linked Lists
- Double Linked Lists
- Circular Double Linked Lists.

NODE:

Each node consists of two fields. Information (info) field and next address (next) field. The info field consists of actual information/data/item that has to be stored in a list. The second field next/link contains the address of the next node. Since next field contains the address,

It is of type pointer. Here the nodes in the list are logically adjacent to each other. Nodes that are physically adjacent need not be logically adjacent in the list.

The entire linked list is accessed from an external pointer FIRST that points to (contains the address of) the first node in the list. (By an “external” pointer, we mean, one that is not included within a node. Rather its value can be accessed directly by referencing a variable).

NODE

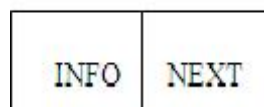


Fig-1 Linked List

The list containing 4 items/data 10, 20, 30 and 40 is shown below.

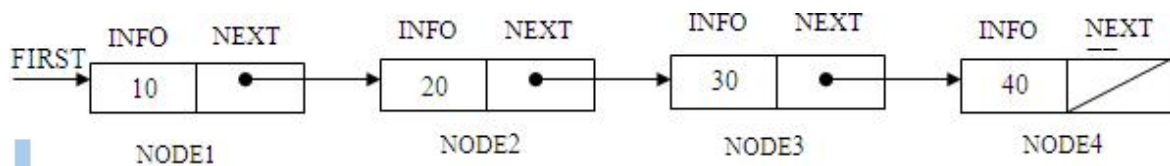


Fig-2 Linked List

The nodes in the list can be accessed using a pointer variable. In the above fig. FIRST is the pointer having the address of the first node of the list, initially before creating the list, as list is empty. The FIRST will always be initialized to NULL in the beginning. Once the list is created, FIRST contains the address of the first node of the list.

As each node is having only one link/next, the list is called single linked list and all the nodes are linked in one direction. Each node can be accessed by the pointer pointing (holding the address) to that node, Say P is pointer to a particular node, then the information field of that node can be accessed using $\text{info}(P)$ and the next field can be accessed using $\text{next}(P)$.

The arrows coming out of the next field in the fig. indicates that the address of the succeeding node is stored in that field. The link field of last node contains a special value known as NULL which is shown using a diagonal line pictorially. This NULL pointer is used to signal the end of a list.

The basic operations of linked lists are Insertion, Deletion and Display. A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and deleted(removed).

The dynamic nature of list may be contrasted with the static nature of an array, whose size remains constant. When an item has to inserted, we will have to create a node, which has to be got from the available free memory of the computer system, So we shall use a mechanism to find an unused node which makes it available to us. For this purpose we shall use the `getnode` operation (`getnode()` function).

The C language provides the built-in functions like `malloc()`, `calloc()`, `realloc()` and `free()`, which are stored in `alloc.h` or `stdlib.h` header files. To dynamically allocate and release the memory locations from/to the computer system.

TREES:

Definition:

A data structure which is accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom. A tree can also be defined as a connected, acyclic di-graph.

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

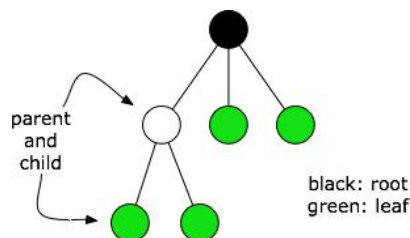


Figure: tree data structure

Fig-3 Tree data structure

Binary tree: A tree with utmost two children for each node.

Complete Binary Tree: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n , the height of the tree, all nodes must be as far left as possible.
Binary search tree: A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

Tree traversal is a technique for processing the nodes of a tree in some order. The different tree traversal techniques are Pre-order, In-order and Post-order traversal. In Pre-order traversal, the tree node is visited first and the left subtree is traversed recursively and later right sub-tree is traversed recursively.

1. Develop a Program in C for the following:

- a. Define a structure Book with the following fields: Book_ID, Title, Author, Price, Availability Status (Available/Issued)
- b. Dynamically allocate memory to store the details of N books, where N is entered by the user.
- c. Implement the following functions:
 - i. create() – To allocate memory and read the details of N books and store them in the allocated memory.
 - ii. display() – To display the details of all books available in the library.
 - iii. search() – To search for a book using Book ID and display the book details if found.
 - iv. issueBook() – To issue a book by updating its availability status and display an appropriate message.
 - v. returnBook() – To return an issued book by updating its availability status and display an appropriate message.
- d. Provide a menu-driven interface with the following options:
 - i. Add Book Records
 - ii. Display All Book Records
 - iii. Search Book by Book ID
 - iv. Issue a Book
 - v. Return a Book
 - vi. Exit

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

// Define the Book structure
struct Book {
    int book_id;
    char title[50];
    char author[50];
    float price;
    char status[20]; // "Available" or "Issued"
};

// Global pointers and variables for dynamic memory management
struct Book *books = NULL;
int n = 0;

// Function declarations
void create();
void display();
void search();
void issueBook();
void returnBook();

int main() {
    int choice;

    do {
        printf("\n--- Library Management System ---\n");
        printf("1. Add Book Records\n");
        printf("2. Display All Book Records\n");
        printf("3. Search Book by Book ID\n");
        printf("4. Issue a Book\n");
        printf("5. Return a Book\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```

switch (choice) {
    case 1:
        create();
        break;
    case 2:
        display();
        break;
    case 3:
        search();
        break;
    case 4:
        issueBook();
        break;
    case 5:
        returnBook();
        break;
    case 6:
        printf("Exiting program. Freeing allocated memory...\n");
        if (books != NULL) {
            free(books);
        }
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 6);

return 0;
}

// i. create() – Allocate memory and read details of N books
void create() {
    printf("Enter the number of books (N): ");
    scanf("%d", &n);

    // Dynamically allocate memory for N books
    books = (struct Book *)malloc(n * sizeof(struct Book));
    if (books == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    for (int i = 0; i < n; i++) {
        printf("\nEnter details for Book %d:\n", i + 1);
        printf("Book ID: ");
        scanf("%d", &books[i].book_id);

        printf("Title: ");
        scanf(" %[^\n]s", books[i].title); // Read string with spaces

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

        printf("Price: ");
        scanf("%f", &books[i].price);
    }
}

```

```

strcpy(books[i].status, "Available");
}
printf("Book records added successfully!\n");
}

// ii. display() – Display details of all books in the library
void display() {
    if (books == NULL || n == 0) {
        printf("\nNo book records found. Please add books first.\n");
        return;
    }

    printf("\n--- Library Books Details ---\n");
    printf("%-10s %-25s %-20s %-10s %-15s\n", "Book ID", "Title", "Author", "Price", "Status");
    printf("-----\n");
    for (int i = 0; i < n; i++) {
        printf("%-10d %-25s %-20s %-10.2f %-15s\n",
            books[i].book_id,
            books[i].title,
            books[i].author,
            books[i].price,
            books[i].status);
    }
}

// iii. search() – Search for a book using Book ID
void search() {
    if (books == NULL || n == 0) {
        printf("\nNo book records available to search.\n");
        return;
    }

    int target_id, found = 0;
    printf("Enter Book ID to search: ");
    scanf("%d", &target_id);

    for (int i = 0; i < n; i++) {
        if (books[i].book_id == target_id) {
            printf("\nBook Found!\n");
            printf("Book ID: %d\n", books[i].book_id);
            printf("Title: %s\n", books[i].title);
            printf("Author: %s\n", books[i].author);
            printf("Price: %.2f\n", books[i].price);
            printf("Status: %s\n", books[i].status);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Book with ID %d not found.\n", target_id);
    }
}

// iv. issueBook() – Issue a book by updating availability status
void issueBook() {
    if (books == NULL || n == 0) {

```

```
printf("\nNo book records available.\n");
return;
}

int target_id, found = 0;
printf("Enter Book ID to issue: ");
scanf("%d", &target_id);

for (int i = 0; i < n; i++) {
    if (books[i].book_id == target_id) {
        found = 1;
        if (strcmp(books[i].status, "Available") == 0) {
            strcpy(books[i].status, "Issued");
            printf("Success! Book '%s' has been issued.\n", books[i].title);
        } else {
            printf("Sorry, the book '%s' is already issued.\n", books[i].title);
        }
        break;
    }
}

if (!found) {
    printf("Book with ID %d not found.\n", target_id);
}
}

// v. returnBook() – Return an issued book by updating availability status
void returnBook() {
    if (books == NULL || n == 0) {
        printf("\nNo book records available.\n");
        return;
    }

    int target_id, found = 0;
    printf("Enter Book ID to return: ");
    scanf("%d", &target_id);

    for (int i = 0; i < n; i++) {
        if (books[i].book_id == target_id) {
            found = 1;
            if (strcmp(books[i].status, "Issued") == 0) {
                strcpy(books[i].status, "Available");
                printf("Success! Book '%s' has been returned and is now available.\n", books[i].title);
            } else {
                printf("Error! The book '%s' was not marked as issued.\n", books[i].title);
            }
            break;
        }
    }

    if (!found) {
        printf("Book with ID %d not found.\n", target_id);
    }
}
```

2. Develop a menu driven Program in C for the following operations on STACK of Integers (Array Implementation of Stack with maximum size MAX)
- Push an Element on to Stack
 - Pop an Element from Stack
 - Demonstrate how Stack can be used to check Palindrome
 - Demonstrate Overflow and Underflow situations on Stack
 - Display the status of Stack
 - Exit

code:

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

#define MAX 5 // Maximum size of the stack

int stack[MAX];
int top = -1;

// Function declarations
void push();
void pop();
void display();
void checkPalindrome();
void checkOverflowUnderflow();

int main() {
    int choice;

    do {
        printf("\n--- Stack Operations Menu ---\n");
        printf("1. Push an Element\n");
        printf("2. Pop an Element\n");
        printf("3. Check Palindrome using Stack\n");
        printf("4. Demonstrate Overflow and Underflow\n");
        printf("5. Display Stack Status\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                checkPalindrome();
                break;
            case 4:
                checkOverflowUnderflow();
                break;
            case 5:
                display();
                break;
            case 6:
                break;
        }
    } while (choice != 6);
}
```

```
        printf("Exiting program...\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
    }
} while (choice != 6);

return 0;
}

// a. Push an Element on to Stack
void push() {
    int element;
    if (top == MAX - 1) {
        printf("\nStack Overflow! Cannot push element, stack is full.\n");
        return;
    }
    printf("Enter the integer element to push: ");
    scanf("%d", &element);
    top++;
    stack[top] = element;
    printf("Successfully pushed %d onto the stack.\n", element);
}

// b. Pop an Element from Stack
void pop() {
    if (top == -1) {
        printf("\nStack Underflow! Cannot pop element, stack is empty.\n");
        return;
    }
    printf("Popped element: %d\n", stack[top]);
    top--;
}

// c. Demonstrate how Stack can be used to check Palindrome
void checkPalindrome() {
    char str[100];
    int length, i;

    // Reset stack temporarily or use a local emulation for string processing
    // To keep it standard, we use the stack array approach for character ASCII values or digits
    printf("Enter a string or number to check for palindrome: ");
    scanf("%s", str);

    length = strlen(str);
    if (length > MAX) {
        printf("Error: String length exceeds maximum stack capacity (%d).\n", MAX);
        return;
    }

    // Push all characters onto the stack
    int temp_top = -1;
    for (i = 0; i < length; i++) {
        temp_top++;
        stack[temp_top] = str[i];
    }
}
```

```

// Pop and compare characters
int isPalindrome = 1;
for (i = 0; i < length; i++) {
    if (str[i] != stack[temp_top]) {
        isPalindrome = 0;
        break;
    }
    temp_top--;
}

if (isPalindrome) {
    printf("Result: '%s' is a PALINDROME.\n", str);
} else {
    printf("Result: '%s' is NOT a palindrome.\n", str);
}
}

// d. Demonstrate Overflow and Underflow situations on Stack
void checkOverflowUnderflow() {
    printf("\n--- Demonstrating Overflow and Underflow ---\n");

    // Save current stack state if needed, or demonstrate conceptually using local tracking
    int demo_stack[MAX];
    int demo_top = -1;

    printf("1. Demonstrating Underflow:\n");
    if (demo_top == -1) {
        printf("  Stack is currently empty (top = -1). Attempting to pop...\n");
        printf("  -> Underflow Encountered! No elements to pop.\n");
    }

    printf("\n2. Demonstrating Overflow:\n");
    printf("  Filling the stack up to maximum size (MAX = %d)...\n", MAX);
    for (int i = 0; i < MAX; i++) {
        demo_top++;
        demo_stack[demo_top] = (i + 1) * 10;
        printf("  Pushed: %d (Current top: %d)\n", demo_stack[demo_top], demo_top);
    }

    // Try pushing one more element
    if (demo_top == MAX - 1) {
        printf("  Stack is full (top = %d). Attempting to push another element...\n", demo_top);
        printf("  -> Overflow Encountered! Cannot push further.\n");
    }
}

// e. Display the status of Stack
void display() {
    if (top == -1) {
        printf("\nStack Status: Stack is EMPTY.\n");
        return;
    }

    printf("\nStack Status: Stack has %d element(s) (Max size: %d).\n", top + 1, MAX);
    printf("Current Stack Elements (Top to Bottom):\n");
    for (int i = top; i >= 0; i--) {
        printf("| %d |\n", stack[i]);
    }
}

```

```

    if (i == top) {
        printf(" ^-- Top\n");
    }
}
printf("-----\n");
}

```

3. Develop a program in C for simulating a Printer Queue using Queue data structure with the following operations:

- a. Add a print job to the Printer Queue.
- b. Process (delete) a print job from the Printer Queue.
- c. Display all pending print jobs in the Queue.
- d. Display the number of print jobs currently waiting in the Queue.
- e. Demonstrate Queue overflow and underflow conditions.
- f. Exit

```

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

```

```

#define MAX 5 // Maximum size of the Printer Queue

```

```

// Structure to represent a Print Job

```

```

struct PrintJob {
    int job_id;
    char document_name[50];
    int pages;
};

```

```

struct PrintJob queue[MAX];
int front = -1;
int rear = -1;

```

```

// Function declarations

```

```

void addJob();
void processJob();
void displayJobs();
void countJobs();
void demonstrateOverflowUnderflow();

```

```

int main() {
    int choice;

```

```

    do {
        printf("\n--- Printer Queue Simulation Menu ---\n");
        printf("1. Add a Print Job\n");
        printf("2. Process (Delete) a Print Job\n");
        printf("3. Display All Pending Print Jobs\n");
        printf("4. Display Number of Waiting Jobs\n");
        printf("5. Demonstrate Queue Overflow and Underflow\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

```

```

        switch (choice) {
            case 1:
                addJob();
                break;

```

```

    case 2:
        processJob();
        break;
    case 3:
        displayJobs();
        break;
    case 4:
        countJobs();
        break;
    case 5:
        demonstrateOverflowUnderflow();
        break;
    case 6:
        printf("Exiting Printer Queue simulation...\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
    }
} while (choice != 6);

return 0;
}

// a. Add a print job to the Printer Queue (Enqueue)
void addJob() {
    // Check for Queue Overflow
    if (rear == MAX - 1) {
        printf("\nQueue Overflow! Printer queue is full. Cannot add more print jobs.\n");
        return;
    }

    struct PrintJob newJob;
    printf("Enter Job ID: ");
    scanf("%d", &newJob.job_id);
    printf("Enter Document Name: ");
    scanf(" %[^\\n]s", newJob.document_name);
    printf("Enter Number of Pages: ");
    scanf("%d", &newJob.pages);

    if (front == -1) {
        front = 0; // First element being inserted
    }
    rear++;
    queue[rear] = newJob;
    printf("Success! Print Job '%s' (ID: %d) added to the queue.\n", newJob.document_name, newJob.job_id);
}

// b. Process (delete) a print job from the Printer Queue (Dequeue)
void processJob() {
    // Check for Queue Underflow
    if (front == -1 || front > rear) {
        printf("\nQueue Underflow! No pending print jobs to process.\n");
        front = -1;
        rear = -1;
        return;
    }
}

```

```

printf("\n--- Processing Print Job ---\n");
printf("Job ID      : %d\n", queue[front].job_id);
printf("Document Name: %s\n", queue[front].document_name);
printf("Pages       : %d\n", queue[front].pages);
printf("Status      : Printed and removed from queue.\n");

front++;

// Reset queue if it becomes empty
if (front > rear) {
    front = -1;
    rear = -1;
}
}

// c. Display all pending print jobs in the Queue
void displayJobs() {
    if (front == -1 || front > rear) {
        printf("\nPrinter Queue is EMPTY. No pending jobs.\n");
        return;
    }

    printf("\n--- Pending Print Jobs in Queue ---\n");
    printf("%-10s %-25s %-10s\n", "Job ID", "Document Name", "Pages");
    printf("-----\n");
    for (int i = front; i <= rear; i++) {
        printf("%-10d %-25s %-10d\n",
            queue[i].job_id,
            queue[i].document_name,
            queue[i].pages);
    }
}

// d. Display the number of print jobs currently waiting in the Queue
void countJobs() {
    if (front == -1 || front > rear) {
        printf("\nNumber of jobs currently waiting: 0\n");
        return;
    }
    int count = (rear - front) + 1;
    printf("\nNumber of print jobs currently waiting in the queue: %d\n", count);
}

// e. Demonstrate Queue overflow and underflow conditions
void demonstrateOverflowUnderflow() {
    printf("\n--- Demonstrating Queue Overflow and Underflow ---\n");

    // Create a temporary queue setup for demonstration
    int demo_front = -1, demo_rear = -1;
    int demo_queue[MAX];

    printf("1. Demonstrating Underflow:\n");
    if (demo_front == -1) {
        printf(" Queue is currently empty (front = -1, rear = -1).\n");
        printf(" Attempting to process/dequeue a job...\n");
        printf(" -> Queue Underflow Encountered! No jobs to process.\n");
    }
}

```

```

printf("\n2. Demonstrating Overflow:\n");
printf(" Filling the printer queue up to maximum size (MAX = %d)...\n", MAX);
for (int i = 0; i < MAX; i++) {
    if (demo_front == -1) demo_front = 0;
    demo_rear++;
    demo_queue[demo_rear] = (i + 101); // Dummy Job ID
    printf(" Added Job ID: %d (Queue rear index: %d)\n", demo_queue[demo_rear], demo_rear);
}

// Try adding one more element when full
if (demo_rear == MAX - 1) {
    printf(" Queue is now full (rear = %d). Attempting to add one more job...\n", demo_rear);
    printf(" -> Queue Overflow Encountered! Cannot add job, queue is full.\n");
}
}

```

4. Develop a menu-driven Program in C for the following operations on a Singly Linked List of Student Records:

- Create a linked list using front insertion.
- Display all student records and count the number of nodes.
- Insert and delete nodes at the beginning and end of the list.
- Search for a student record using USN.

```

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

```

// Define the Node structure for Student Record

```

struct Student {
    char usn[20];
    char name[50];
    char branch[20];
    int sem;
    char phone[15];
    struct Student *next;
};

```

// Global pointer to the head of the linked list

```

struct Student *head = NULL;

```

// Function declarations

```

void createFront();
void displayAndCount();
void insertBegin();
void insertEnd();
void deleteBegin();
void deleteEnd();
void searchUSN();

```

```

int main() {
    int choice;

    do {
        printf("\n--- Singly Linked List - Student Management System ---\n");
        printf("1. Create Linked List (Front Insertion)\n");
        printf("2. Display All Records & Count Nodes\n");
        printf("3. Insert at Beginning\n");
    } while (choice != 4);
}

```

```

printf("4. Insert at End\n");
printf("5. Delete from Beginning\n");
printf("6. Delete from End\n");
printf("7. Search Student by USN\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        createFront();
        break;
    case 2:
        displayAndCount();
        break;
    case 3:
        insertBegin();
        break;
    case 4:
        insertEnd();
        break;
    case 5:
        deleteBegin();
        break;
    case 6:
        deleteEnd();
        break;
    case 7:
        searchUSN();
        break;
    case 8:
        printf("Exiting program. Freeing memory...\n");
        // Free all remaining nodes before exit
        struct Student *temp;
        while (head != NULL) {
            temp = head;
            head = head->next;
            free(temp);
        }
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 8);

return 0;
}

// Helper function to read student details
struct Student* createNode() {
    struct Student *newNode = (struct Student *)malloc(sizeof(struct Student));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    printf("Enter USN: ");
    scanf("%s", newNode->usn);

```

```

printf("Enter Name: ");
scanf(" %[^\\n]s", newNode->name);
printf("Enter Branch: ");
scanf("%s", newNode->branch);
printf("Enter Semester: ");
scanf("%d", &newNode->sem);
printf("Enter Phone Number: ");
scanf("%s", newNode->phone);
newNode->next = NULL;
return newNode;
}

// a. Create a linked list using front insertion
void createFront() {
    int n;
    printf("Enter the number of student records to create: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("\\nEnter details for Student %d:\\n", i + 1);
        struct Student *newNode = createNode();
        newNode->next = head;
        head = newNode;
        printf("Student record inserted at the front successfully.\\n");
    }
}

// b. Display all student records and count the number of nodes
void displayAndCount() {
    if (head == NULL) {
        printf("\\nStudent list is EMPTY. No records to display.\\n");
        return;
    }

    struct Student *temp = head;
    int count = 0;

    printf("\\n--- Student Records List ---\\n");
    printf("%-15s %-20s %-10s %-5s %-15s\\n", "USN", "Name", "Branch", "Sem", "Phone");
    printf("-----\\n");

    while (temp != NULL) {
        printf("%-15s %-20s %-10s %-5d %-15s\\n",
            temp->usn,
            temp->name,
            temp->branch,
            temp->sem,
            temp->phone);
        count++;
        temp = temp->next;
    }
    printf("-----\\n");
    printf("Total number of student records (nodes): %d\\n", count);
}

// c. Insert node at the beginning
void insertBegin() {

```

```
printf("\nEnter details for the new student (to insert at beginning):\n");
struct Student *newNode = createNode();
newNode->next = head;
head = newNode;
printf("Student inserted at the beginning successfully.\n");
}
```

// c. Insert node at the end

```
void insertEnd() {
    printf("\nEnter details for the new student (to insert at end):\n");
    struct Student *newNode = createNode();

    if (head == NULL) {
        head = newNode;
    } else {
        struct Student *temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Student inserted at the end successfully.\n");
}
```

// c. Delete node from the beginning

```
void deleteBegin() {
    if (head == NULL) {
        printf("\nList Underflow! No student records to delete.\n");
        return;
    }

    struct Student *temp = head;
    head = head->next;
    printf("\nDeleted student record from beginning:\n");
    printf("USN: %s | Name: %s\n", temp->usn, temp->name);
    free(temp);
}
```

// c. Delete node from the end

```
void deleteEnd() {
    if (head == NULL) {
        printf("\nList Underflow! No student records to delete.\n");
        return;
    }

    struct Student *temp = head;
    struct Student *prev = NULL;

    // If there is only one node in the list
    if (head->next == NULL) {
        printf("\nDeleted student record from end:\n");
        printf("USN: %s | Name: %s\n", head->usn, head->name);
        free(head);
        head = NULL;
        return;
    }
```

```

// Traverse until the last node
while (temp->next != NULL) {
    prev = temp;
    temp = temp->next;
}

prev->next = NULL;
printf("\nDeleted student record from end:\n");
printf("USN: %s | Name: %s\n", temp->usn, temp->name);
free(temp);
}

// d. Search for a student record using USN
void searchUSN() {
    if (head == NULL) {
        printf("\nList is empty. Cannot search.\n");
        return;
    }

    char target_usn[20];
    printf("Enter USN to search: ");
    scanf("%s", target_usn);

    struct Student *temp = head;
    int found = 0;

    while (temp != NULL) {
        if (strcmp(temp->usn, target_usn) == 0) {
            printf("\nStudent Record Found!\n");
            printf("USN : %s\n", temp->usn);
            printf("Name : %s\n", temp->name);
            printf("Branch: %s\n", temp->branch);
            printf("Sem : %d\n", temp->sem);
            printf("Phone : %s\n", temp->phone);
            found = 1;
            break;
        }
        temp = temp->next;
    }

    if (!found) {
        printf("Student with USN '%s' not found in the list.\n", target_usn);
    }
}

```

5. Develop a Program in C for the following:

a. Create a Binary Tree by inserting integer elements entered by the user in level order.

b. Implement the following functions:

i. createTree() – Create a binary tree using dynamic memory allocation.

ii. preorder() – Display the elements using Preorder Traversal.

iii. inorder() – Display the elements using Inorder Traversal.

iv. postorder() – Display the elements using Postorder Traversal.

c. Provide a menu-driven interface with the following options:

i. Create Binary Tree

ii. Display Preorder Traversal

iii. Display Inorder Traversal

iv. Display Postorder Traversal

v. Exit

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

// Define structure for a Binary Tree Node
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

// Define structure for a Queue Node (used for level-order creation)
struct QueueNode {
    struct Node *treeNode;
    struct QueueNode *next;
};

// Queue pointers for level order insertion
struct QueueNode *front = NULL;
struct QueueNode *rear = NULL;

// Global root of the binary tree
struct Node *root = NULL;

// Function declarations
struct Node* createNode(int value);
void enqueue(struct Node *node);
struct Node* dequeue();
void createTree();
void preorder(struct Node *curr);
void inorder(struct Node *curr);
void postorder(struct Node *curr);

int main() {
    int choice;

    do {
        printf("\n--- Binary Tree Operations Menu ---\n");
        printf("i. Create Binary Tree\n");
        printf("ii. Display Preorder Traversal\n");
        printf("iii. Display Inorder Traversal\n");
        printf("iv. Display Postorder Traversal\n");
        printf("v. Exit\n");
        printf("Enter your choice (1-5): ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                createTree();
                break;
            case 2:
                printf("\nPreorder Traversal: ");
                if (root == NULL) {
                    printf("Tree is empty.");
                } else {
                    preorder(root);
                }
                printf("\n");
            default:
                break;
        }
    } while (choice != 5);
}
```

```

        break;
    case 3:
        printf("\nInorder Traversal: ");
        if (root == NULL) {
            printf("Tree is empty.");
        } else {
            inorder(root);
        }
        printf("\n");
        break;
    case 4:
        printf("\nPostorder Traversal: ");
        if (root == NULL) {
            printf("Tree is empty.");
        } else {
            postorder(root);
        }
        printf("\n");
        break;
    case 5:
        printf("Exiting program...\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
    }
} while (choice != 5);

return 0;
}

// Helper: Create a new tree node using dynamic memory allocation
struct Node* createNode(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Queue Helper: Enqueue tree node pointer
void enqueue(struct Node *node) {
    struct QueueNode *qNode = (struct QueueNode *)malloc(sizeof(struct QueueNode));
    qNode->treeNode = node;
    qNode->next = NULL;
    if (rear == NULL) {
        front = rear = qNode;
    } else {
        rear->next = qNode;
        rear = qNode;
    }
}

// Queue Helper: Dequeue tree node pointer

```

```

struct Node* dequeue() {
    if (front == NULL) return NULL;
    struct QueueNode *temp = front;
    struct Node *node = temp->treeNode;
    front = front->next;
    if (front == NULL) rear = NULL;
    free(temp);
    return node;
}

// i. createTree() – Create binary tree by inserting elements in level order
void createTree() {
    int n, val;
    printf("Enter the number of nodes to insert: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Invalid number of nodes.\n");
        return;
    }

    // Reset root and queue for a fresh tree creation
    root = NULL;
    front = rear = NULL;

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &val);

        struct Node *newNode = createNode(val);

        if (root == NULL) {
            root = newNode;
        } else {
            struct Node *curr = front->treeNode;

            if (curr->left == NULL) {
                curr->left = newNode;
            } else if (curr->right == NULL) {
                curr->right = newNode;
                dequeue(); // Parent has both children filled, remove from queue
            }
        }
        enqueue(newNode);
    }

    // Clear leftover queue elements
    front = rear = NULL;
    printf("Binary Tree created successfully using level-order insertion!\n");
}

// ii. preorder() – Display elements using Preorder Traversal (Root, Left, Right)
void preorder(struct Node *curr) {
    if (curr != NULL) {
        printf("%d ", curr->data);
        preorder(curr->left);
        preorder(curr->right);
    }
}

```

```
}
}
```

// iii. inorder() – Display elements using Inorder Traversal (Left, Root, Right)

```
void inorder(struct Node *curr) {
    if (curr != NULL) {
        inorder(curr->left);
        printf("%d ", curr->data);
        inorder(curr->right);
    }
}
```

// iv. postorder() – Display elements using Postorder Traversal (Left, Right, Root)

```
void postorder(struct Node *curr) {
    if (curr != NULL) {
        postorder(curr->left);
        postorder(curr->right);
        printf("%d ", curr->data);
    }
}
```

6. Develop a Program in C for the following operations on Graph(G) of Cities

a. Create an Adjacency Matrix representing the graph of N cities

b. Print all the nodes reachable from a given starting node in a digraph using DFS/BFS method.

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

```
#define MAX 20
```

```
// Function declarations
```

```
void createGraph(int adj[MAX][MAX], int n);
void printAdjacencyMatrix(int adj[MAX][MAX], int n);
void bfs(int adj[MAX][MAX], int n, int start);
void dfs(int adj[MAX][MAX], int n, int start, int visited[]);
```

```
int main() {
    int adj[MAX][MAX];
    int n, start, choice;

    printf("Enter the number of cities (N): ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX) {
        printf("Invalid number of cities. Must be between 1 and %d.\n", MAX);
        return 1;
    }
}
```

// a. Create an Adjacency Matrix representing the graph of N cities

```
createGraph(adj, n);
```

```
do {
    printf("\n--- Graph Operations Menu ---\n");
    printf("1. Display Adjacency Matrix\n");
    printf("2. Find Reachable Nodes using BFS (Breadth-First Search)\n");
    printf("3. Find Reachable Nodes using DFS (Depth-First Search)\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
}
```

```

switch (choice) {
    case 1:
        printAdjacencyMatrix(adj, n);
        break;
    case 2:
        printf("Enter the starting city index (0 to %d): ", n - 1);
        scanf("%d", &start);
        if (start < 0 || start >= n) {
            printf("Invalid starting city index!\n");
        } else {
            bfs(adj, n, start);
        }
        break;
    case 3: {
        printf("Enter the starting city index (0 to %d): ", n - 1);
        scanf("%d", &start);
        if (start < 0 || start >= n) {
            printf("Invalid starting city index!\n");
        } else {
            int visited[MAX] = {0};
            printf("\nNodes reachable from city %d using DFS: ", start);
            dfs(adj, n, start, visited);
            printf("\n");
        }
        break;
    }
    case 4:
        printf("Exiting program...\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 4);

return 0;
}

// a. Create an Adjacency Matrix representing the graph
void createGraph(int adj[MAX][MAX], int n) {
    printf("\nEnter the adjacency matrix elements (%d x %d):\n", n, n);
    printf("(Enter 1 if there is a directed edge from row to column, otherwise 0)\n");

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("Edge from city %d to city %d (0/1): ", i, j);
            scanf("%d", &adj[i][j]);
        }
    }
    printf("Adjacency Matrix created successfully!\n");
}

// Helper: Print Adjacency Matrix
void printAdjacencyMatrix(int adj[MAX][MAX], int n) {
    printf("\n--- Adjacency Matrix ---\n");
    for (int i = 0; i < n; i++) {
        printf("C%d ", i);
    }

```

```

}
printf("\n-----\n");
for (int i = 0; i < n; i++) {
    printf("C%d | ", i);
    for (int j = 0; j < n; j++) {
        printf("%d ", adj[i][j]);
    }
    printf("\n");
}
}

// b. BFS Method to print all reachable nodes
void bfs(int adj[MAX][MAX], int n, int start) {
    int queue[MAX], front = 0, rear = -1;
    int visited[MAX] = {0};

    printf("\nNodes reachable from city %d using BFS: ", start);

    visited[start] = 1;
    queue[++rear] = start;

    while (front <= rear) {
        int curr = queue[front++];
        printf("C%d ", curr);

        for (int i = 0; i < n; i++) {
            if (adj[curr][i] == 1 && !visited[i]) {
                visited[i] = 1;
                queue[++rear] = i;
            }
        }
    }
    printf("\n");
}

```

```

// b. DFS Method to print all reachable nodes
void dfs(int adj[MAX][MAX], int n, int start, int visited[]) {
    visited[start] = 1;
    printf("C%d ", start);

    for (int i = 0; i < n; i++) {
        if (adj[start][i] == 1 && !visited[i]) {
            dfs(adj, n, i, visited);
        }
    }
}

```

7. Develop a C program to:

- a. Read two sparse matrices of the same order.
- b. Convert both into 3-tuple representation.
- c. Perform matrix addition.
- d. Display the resultant matrix in both normal matrix form and 3-tuple form.

```

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

```

```

#define MAX 100

```

```
// Structure to represent a term in 3-tuple format
struct SparseTuple {
    int row;
    int col;
    int value;
};

// Function declarations
void readSparseMatrix(int matrix[MAX][MAX], int rows, int cols);
void convertToTuple(int matrix[MAX][MAX], int rows, int cols, struct SparseTuple tuple[]);
void addTuple(struct SparseTuple t1[], struct SparseTuple t2[], struct SparseTuple result[]);
void displayTuple(struct SparseTuple tuple[]);
void displayNormalMatrix(struct SparseTuple tuple[]);

int main() {
    int rows, cols;
    int matrix1[MAX][MAX], matrix2[MAX][MAX];
    struct SparseTuple t1[MAX], t2[MAX], result[MAX];

    printf("Enter the number of rows and columns for the sparse matrices: ");
    scanf("%d %d", &rows, &cols);

    // a. Read two sparse matrices of the same order
    printf("\nEnter elements for Matrix 1 (%dx%d):\n", rows, cols);
    readSparseMatrix(matrix1, rows, cols);

    printf("\nEnter elements for Matrix 2 (%dx%d):\n", rows, cols);
    readSparseMatrix(matrix2, rows, cols);

    // b. Convert both into 3-tuple representation
    t1[0].row = rows;
    t1[0].col = cols;
    convertToTuple(matrix1, rows, cols, t1);

    t2[0].row = rows;
    t2[0].col = cols;
    convertToTuple(matrix2, rows, cols, t2);

    // Display the initial 3-tuple representations
    printf("\n--- 3-Tuple Representation of Matrix 1 ---");
    displayTuple(t1);

    printf("\n--- 3-Tuple Representation of Matrix 2 ---");
    displayTuple(t2);

    // c. Perform matrix addition using 3-tuple representation
    addTuple(t1, t2, result);

    // d. Display the resultant matrix in both 3-tuple form and normal matrix form
    printf("\n--- Resultant Matrix (3-Tuple Form) ---");
    displayTuple(result);

    printf("\n--- Resultant Matrix (Normal Form) ---");
    displayNormalMatrix(result);

    return 0;
}
```

```

// Function to read a standard matrix from user input
void readSparseMatrix(int matrix[MAX][MAX], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Element [%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }
}

// Function to convert a normal matrix into 3-tuple format
void convertToTuple(int matrix[MAX][MAX], int rows, int cols, struct SparseTuple tuple[]) {
    int k = 1; // Index for non-zero elements in tuple array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] != 0) {
                tuple[k].row = i;
                tuple[k].col = j;
                tuple[k].value = matrix[i][j];
                k++;
            }
        }
    }
    tuple[0].value = k - 1; // Total number of non-zero values stored at index 0
}

// Function to add two sparse matrices given in 3-tuple form
void addTuple(struct SparseTuple t1[], struct SparseTuple t2[], struct SparseTuple result[]) {
    // Check if matrix dimensions match
    if (t1[0].row != t2[0].row || t1[0].col != t2[0].col) {
        printf("\nError: Matrix orders must be the same for addition!\n");
        result[0].row = 0;
        result[0].col = 0;
        result[0].value = 0;
        return;
    }

    result[0].row = t1[0].row;
    result[0].col = t1[0].col;

    int i = 1, j = 1, k = 1;

    while (i <= t1[0].value && j <= t2[0].value) {
        // Compare row indices first
        if (t1[i].row < t2[j].row || (t1[i].row == t2[j].row && t1[i].col < t2[j].col)) {
            result[k] = t1[i];
            i++;
            k++;
        }
        // Compare column indices if rows are equal
        else if (t2[j].row < t1[i].row || (t2[j].row == t1[i].row && t2[j].col < t1[i].col)) {
            result[k] = t2[j];
            j++;
            k++;
        }
        // Both row and column match, add their values
    }
}

```

```

else {
    int sum = t1[i].value + t2[j].value;
    if (sum != 0) {
        result[k].row = t1[i].row;
        result[k].col = t1[i].col;
        result[k].value = sum;
        k++;
    }
    i++;
    j++;
}
}

// Copy remaining elements from t1, if any
while (i <= t1[0].value) {
    result[k++] = t1[i++];
}

// Copy remaining elements from t2, if any
while (j <= t2[0].value) {
    result[k++] = t2[j++];
}

result[0].value = k - 1; // Set total non-zero count
}

// Function to display a 3-tuple representation table
void displayTuple(struct SparseTuple tuple[]) {
    int total_elements = tuple[0].value;
    printf("\n%-8s %-8s %-8s\n", "Row", "Column", "Value");
    printf("-----\n");
    printf("%-8d %-8d %-8d (Header)\n", tuple[0].row, tuple[0].col, tuple[0].value);
    for (int i = 1; i <= total_elements; i++) {
        printf("%-8d %-8d %-8d\n", tuple[i].row, tuple[i].col, tuple[i].value);
    }
}

// Function to display the 3-tuple format back into normal matrix form
void displayNormalMatrix(struct SparseTuple tuple[]) {
    int rows = tuple[0].row;
    int cols = tuple[0].col;
    int total = tuple[0].value;
    int k = 1;

    printf("\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (k <= total && tuple[k].row == i && tuple[k].col == j) {
                printf("%-4d ", tuple[k].value);
                k++;
            } else {
                printf("%-4d ", 0);
            }
        }
        printf("\n");
    }
}

```

8. Develop a C-based Expression Conversion tool that uses stack data structure to convert arithmetic expressions from infix notation to postfix notation. The tool should support alphanumeric operands, parentheses, and the operators (+, -, *, /, %, ^). Analyze the correctness of the conversion by testing various valid expressions.

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

#define MAX 100

// Stack structure for operators
char stack[MAX];
int top = -1;

// Stack helper functions
void push(char item) {
    if (top >= MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = item;
    }
}

char pop() {
    if (top < 0) {
        return -1;
    } else {
        return stack[top--];
    }
}

char peek() {
    if (top < 0) {
        return -1;
    } else {
        return stack[top];
    }
}

// Function to determine precedence of operators
int precedence(char symbol) {
    switch (symbol) {
        case '^':
            return 3;
        case '*':
        case '/':
        case '%':
            return 2;
        case '+':
        case '-':
            return 1;
        default:
            return 0;
    }
}
```

}

```
// Function to check associativity (Right-to-Left for '^', Left-to-Right otherwise)
```

```
int isRightAssociative(char symbol) {
    if (symbol == '^')
        return 1;
    return 0;
}
```

```
// Function to convert infix expression to postfix notation
```

```
void infixToPostfix(char infix[], char postfix[]) {
    int i, j = 0;
    char symbol;

    top = -1; // Reset stack

    for (i = 0; infix[i] != '\0'; i++) {
        symbol = infix[i];

        // Skip whitespace if any
        if (symbol == ' ' || symbol == '\t')
            continue;

        // If the scanned character is an operand (alphanumeric), add it to output
        if (isalnum(symbol)) {
            postfix[j++] = symbol;
        }
        // If the scanned character is '(', push it to the stack
        else if (symbol == '(') {
            push(symbol);
        }
        // If the scanned character is ')', pop and output from the stack until '(' is encountered
        else if (symbol == ')') {
            while (top != -1 && peek() != '(') {
                postfix[j++] = pop();
            }
            if (top != -1 && peek() == '(') {
                pop(); // Remove the '(' from stack
            }
        }
        // An operator is encountered
        else {
            while (top != -1 && peek() != '(' &&
                (precedence(peek()) > precedence(symbol) ||
                (precedence(peek()) == precedence(symbol) && !isRightAssociative(symbol)))) {
                postfix[j++] = pop();
            }
            push(symbol);
        }
    }
}
```

```
// Pop all remaining elements from the stack
```

```
while (top != -1) {
    postfix[j++] = pop();
}
```

```
postfix[j] = '\0'; // Null-terminate the postfix string
```

}

// Function to run test cases for verifying correctness

```
void runTestCases() {
    char *testCases[][2] = {
        {"A+B*C", "ABC*+"},
        {"(A+B)*C", "AB+C*"},
        {"A^B^C", "AB^C^"},
        {"A*(B+C)/D", "ABC+*D/"},
        {"a+b*(c-d)/e", "abcd-*e/+/%"}, // Wait, let's keep valid standard ones:
        {"A+B*C-D/E", "ABC*+DE/-"},
        {"(A+B)*(C-D)", "AB+CD-*"}
    };
};
```

```
int numTests = sizeof(testCases) / sizeof(testCases[0]);
```

```
printf("\n--- Running Validation Test Cases ---\n");
```

```
for (int i = 0; i < numTests; i++) {
    char postfix[MAX];
    infixToPostfix(testCases[i][0], postfix);
    printf("Test %d:\n", i + 1);
    printf(" Infix : %s\n", testCases[i][0]);
    printf(" Postfix: %s\n\n", postfix);
}
}
```

```
int main() {
```

```
    int choice;
```

```
    char infix[MAX], postfix[MAX];
```

```
do {
```

```
    printf("\n--- Expression Conversion Tool (Infix to Postfix) ---\n");
```

```
    printf("1. Convert Infix Expression to Postfix\n");
```

```
    printf("2. Run Built-in Validation Test Cases\n");
```

```
    printf("3. Exit\n");
```

```
    printf("Enter your choice: ");
```

```
    scanf("%d", &choice);
```

```
    getchar(); // Clear newline buffer
```

```
switch (choice) {
```

```
    case 1:
```

```
        printf("Enter a valid infix expression: ");
```

```
        fgets(infix, sizeof(infix), stdin);
```

```
        // Remove trailing newline if present
```

```
        infix[strcspn(infix, "\n")] = 0;
```

```
        infixToPostfix(infix, postfix);
```

```
        printf("Postfix Expression: %s\n", postfix);
```

```
        break;
```

```
    case 2:
```

```
        runTestCases();
```

```
        break;
```

```
    case 3:
```

```
        printf("Exiting tool...\n");
```

```
        break;
```

```
    default:
```

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

```
}
```

```
} while (choice != 3);

return 0;
}
```

9. Develop a menu-driven Program in C for the following operations on Circular Queue (integer/floating point data):

- a. Insert an element into the Circular Queue.
- b. Delete an element from the Circular Queue.
- c. Display the contents of the Circular Queue.
- d. Demonstrate overflow and underflow conditions.
- e. Exit.

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

```
#define MAX 5 // Maximum size of the Circular Queue
```

```
float cqueue[MAX];
int front = -1;
int rear = -1;
```

```
// Function declarations
void insertElement();
void deleteElement();
void displayQueue();
void demonstrateOverflowUnderflow();
```

```
int main() {
    int choice;

    do {
        printf("\n--- Circular Queue Operations Menu ---\n");
        printf("1. Insert an Element\n");
        printf("2. Delete an Element\n");
        printf("3. Display Circular Queue Contents\n");
        printf("4. Demonstrate Overflow and Underflow\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insertElement();
                break;
            case 2:
                deleteElement();
                break;
            case 3:
                displayQueue();
                break;
            case 4:
                demonstrateOverflowUnderflow();
                break;
            case 5:
                printf("Exiting Circular Queue program...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 5);
}
```

```
}
} while (choice != 5);

return 0;
}

// a. Insert an element into the Circular Queue
void insertElement() {
    float element;

    // Check for Queue Overflow
    if ((front == 0 && rear == MAX - 1) || (front == rear + 1)) {
        printf("\nCircular Queue Overflow! Cannot insert element, queue is full.\n");
        return;
    }

    printf("Enter the floating-point/integer element to insert: ");
    scanf("%f", &element);

    if (front == -1) { // If queue is initially empty
        front = 0;
        rear = 0;
    } else if (rear == MAX - 1 && front != 0) {
        rear = 0; // Wrap around to the beginning
    } else {
        rear++;
    }

    cqueue[rear] = element;
    printf("Successfully inserted %.2f into the circular queue.\n", element);
}

// b. Delete an element from the Circular Queue
void deleteElement() {
    // Check for Queue Underflow
    if (front == -1) {
        printf("\nCircular Queue Underflow! Cannot delete element, queue is empty.\n");
        return;
    }

    float deletedItem = cqueue[front];
    printf("Deleted element: %.2f\n", deletedItem);

    if (front == rear) { // Queue has only one element, reset after deletion
        front = -1;
        rear = -1;
    } else if (front == MAX - 1) {
        front = 0; // Wrap around to the beginning
    } else {
        front++;
    }
}

// c. Display the contents of the Circular Queue
void displayQueue() {
    if (front == -1) {
        printf("\nCircular Queue is EMPTY.\n");
    }
}
```

```

    return;
}

printf("\n--- Circular Queue Contents ---\n");
printf("Front index = %d, Rear index = %d\n", front, rear);
printf("Queue elements: ");

if (rear >= front) {
    for (int i = front; i <= rear; i++) {
        printf("%.2f ", cqueue[i]);
    }
} else {
    for (int i = front; i < MAX; i++) {
        printf("%.2f ", cqueue[i]);
    }
    for (int i = 0; i <= rear; i++) {
        printf("%.2f ", cqueue[i]);
    }
}
printf("\n-----\n");
}

// d. Demonstrate overflow and underflow conditions
void demonstrateOverflowUnderflow() {
    printf("\n--- Demonstrating Overflow and Underflow ---\n");

    // Setup temporary variables for simulation
    float demo_q[MAX];
    int d_front = -1, d_rear = -1;

    printf("\n1. Demonstrating Underflow:\n");
    if (d_front == -1) {
        printf(" Queue is currently empty (front = -1, rear = -1).\n");
        printf(" Attempting to delete/dequeue an element...\n");
        printf(" -> Circular Queue Underflow Encountered! No elements to delete.\n");
    }

    printf("\n2. Demonstrating Overflow:\n");
    printf(" Filling the circular queue up to maximum size (MAX = %d)...\n", MAX);

    for (int i = 0; i < MAX; i++) {
        if (d_front == -1) d_front = 0;
        d_rear = (d_rear + 1) % MAX;
        demo_q[d_rear] = (i + 1) * 5.5;
        printf(" Inserted: %.2f (front: %d, rear: %d)\n", demo_q[d_rear], d_front, d_rear);
    }

    // Try inserting one more element when full
    if ((d_front == 0 && d_rear == MAX - 1) || (d_front == d_rear + 1)) {
        int next_rear = (d_rear + 1) % MAX;
        printf(" Queue is full. Attempting to insert another element at rear = %d...\n", next_rear);
        printf(" -> Circular Queue Overflow Encountered! Cannot insert, queue is full.\n");
    }
}

```

10. Develop a menu driven Program in C for the following operations on Doubly Linked List (DLL) of Employee Data with the fields: SSN, Name, Dept, Designation, Sal, PhNo etc.

- a. Create a DLL of N Employees Data by using end insertion.
- b. Display the status of DLL and count the number of nodes in it
- c. Perform Insertion and Deletion at End of DLL
- d. Perform Insertion and Deletion at Front of DLL
- e. Demonstrate how this DLL can be used as Double Ended Queue.
- f. Exit

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

```
// Define the Node structure for Employee Record
```

```
struct Employee {
    char ssn[20];
    char name[50];
    char dept[20];
    char designation[30];
    float sal;
    char phno[15];
    struct Employee *prev;
    struct Employee *next;
};
```

```
// Global pointers for the head and tail of the Doubly Linked List
```

```
struct Employee *head = NULL;
struct Employee *tail = NULL;
```

```
// Function declarations
```

```
struct Employee* createNode();
void createDLL();
void displayAndCount();
void insertEnd();
void deleteEnd();
void insertFront();
void deleteFront();
void demonstrateDeque();
```

```
int main() {
    int choice;
```

```
    do {
        printf("\n--- Doubly Linked List (Employee Management) Menu ---\n");
        printf("1. Create DLL of N Employees (End Insertion)\n");
        printf("2. Display Status & Count Nodes\n");
        printf("3. Insert & Delete at End\n");
        printf("4. Insert & Delete at Front\n");
        printf("5. Demonstrate as Double Ended Queue (Deque)\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                createDLL();
                break;
```

```

    case 2:
        displayAndCount();
        break;
    case 3: {
        int subChoice;
        printf("\n--- End Operations ---\n");
        printf("1. Insert at End\n");
        printf("2. Delete from End\n");
        printf("Enter choice: ");
        scanf("%d", &subChoice);
        if (subChoice == 1) insertEnd();
        else if (subChoice == 2) deleteEnd();
        else printf("Invalid sub-choice!\n");
        break;
    }
    case 4: {
        int subChoice;
        printf("\n--- Front Operations ---\n");
        printf("1. Insert at Front\n");
        printf("2. Delete from Front\n");
        printf("Enter choice: ");
        scanf("%d", &subChoice);
        if (subChoice == 1) insertFront();
        else if (subChoice == 2) deleteFront();
        else printf("Invalid sub-choice!\n");
        break;
    }
    case 5:
        demonstrateDeque();
        break;
    case 6: {
        printf("Exiting program. Freeing memory...\n");
        struct Employee *temp;
        while (head != NULL) {
            temp = head;
            head = head->next;
            free(temp);
        }
        tail = NULL;
        break;
    }
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 6);

return 0;
}

// Helper: Allocate and read details for a new employee node
struct Employee* createNode() {
    struct Employee *newNode = (struct Employee *)malloc(sizeof(struct Employee));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    printf("Enter SSN: ");

```

```

scanf("%s", newNode->ssn);
printf("Enter Name: ");
scanf(" %[^\\n]s", newNode->name);
printf("Enter Department: ");
scanf("%s", newNode->dept);
printf("Enter Designation: ");
scanf(" %[^\\n]s", newNode->designation);
printf("Enter Salary: ");
scanf("%f", &newNode->sal);
printf("Enter Phone Number: ");
scanf("%s", newNode->phno);
newNode->prev = NULL;
newNode->next = NULL;
return newNode;
}

// a. Create a DLL of N Employees Data by using end insertion
void createDLL() {
    int n, i;
    printf("Enter the number of employees (N): ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("\\nEnter details for Employee %d:\\n", i + 1);
        struct Employee *newNode = createNode();

        if (head == NULL) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
    printf("\\nSuccessfully created DLL with %d employee record(s) using end insertion.\\n", n);
}

// b. Display the status of DLL and count the number of nodes in it
void displayAndCount() {
    if (head == NULL) {
        printf("\\nDoubly Linked List is EMPTY. No employee records found.\\n");
        return;
    }

    struct Employee *temp = head;
    int count = 0;

    printf("\\n--- Employee Records Status (DLL) ---\\n");
    printf("%-12s %-20s %-10s %-15s %-10s %-12s\\n", "SSN", "Name", "Dept", "Designation", "Salary", "Phone");
    printf("-----\\n");

    while (temp != NULL) {
        printf("%-12s %-20s %-10s %-15s %-10.2f %-12s\\n",
            temp->ssn,
            temp->name,
            temp->dept,
            temp->designation,
            temp->sal,
            temp->phno);
        temp = temp->next;
    }
}

```

```

        temp->sal,
        temp->phno);
    count++;
    temp = temp->next;
}
printf("-----\n");
printf("Total number of nodes (employees) in DLL: %d\n", count);
}

```

// c. Perform Insertion at End of DLL

```

void insertEnd() {
    printf("\nEnter details of the employee to insert at the END:\n");
    struct Employee *newNode = createNode();

    if (head == NULL) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    printf("Employee record inserted at the end successfully.\n");
}

```

// c. Perform Deletion at End of DLL

```

void deleteEnd() {
    if (head == NULL) {
        printf("\nList Underflow! No employee records to delete.\n");
        return;
    }

    struct Employee *temp = tail;
    printf("\nDeleted employee record from the END:\n");
    printf("SSN: %s | Name: %s\n", temp->ssn, temp->name);

    if (head == tail) { // Only one node in the list
        head = tail = NULL;
    } else {
        tail = tail->prev;
        tail->next = NULL;
    }
    free(temp);
}

```

// d. Perform Insertion at Front of DLL

```

void insertFront() {
    printf("\nEnter details of the employee to insert at the FRONT:\n");
    struct Employee *newNode = createNode();

    if (head == NULL) {
        head = tail = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Employee record inserted at the front successfully.\n");
}

```

```

// d. Perform Deletion at Front of DLL
void deleteFront() {
    if (head == NULL) {
        printf("\nList Underflow! No employee records to delete.\n");
        return;
    }

    struct Employee *temp = head;
    printf("\nDeleted employee record from the FRONT:\n");
    printf("SSN: %s | Name: %s\n", temp->:ssn, temp->name);

    if (head == tail) { // Only one node in the list
        head = tail = NULL;
    } else {
        head = head->next;
        head->prev = NULL;
    }
    free(temp);
}

// e. Demonstrate how this DLL can be used as Double Ended Queue (Deque)
void demonstrateDeque() {
    printf("\n===== \n");
    printf("DEMONSTRATION: Double Ended Queue (Deque) using DLL\n");
    printf("===== \n");
    printf("A Deque allows insertion and deletion at both ends (Front and Rear).\n");
    printf("In our Doubly Linked List implementation:\n");
    printf("- 'Insert at Front' and 'Delete from Front' act on the Front of the Deque.\n");
    printf("- 'Insert at End' and 'Delete from End' act on the Rear of the Deque.\n");
    printf("This provides O(1) time complexity modifications at both ends.\n");
    printf("===== \n");
}

```

11. Develop a menu driven Program in C for the following operations on Binary Search Tree (BST) of Integers.

- a. Create a BST of N Integers
- b. Traverse the BST in Inorder, Preorder and Post Order
- c. Search the BST for a given element (KEY) and report the appropriate message
- d. Exit

```

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

// Define structure for a BST Node
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

// Function declarations
struct Node* insert(struct Node *root, int value);
void inorder(struct Node *root);
void preorder(struct Node *root);
void postorder(struct Node *root);
struct Node* search(struct Node *root, int key);

```

```
int main() {
    struct Node *root = NULL;
    int choice, n, val, key;

    do {
        printf("\n--- Binary Search Tree (BST) Operations Menu ---\n");
        printf("1. Create BST of N Integers\n");
        printf("2. Traverse BST (Inorder, Preorder, Postorder)\n");
        printf("3. Search for a Key in BST\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the number of integers (N): ");
                scanf("%d", &n);
                root = NULL; // Reset root for fresh creation if desired
                printf("Enter %d integers:\n", n);
                for (int i = 0; i < n; i++) {
                    scanf("%d", &val);
                    root = insert(root, val);
                }
                printf("BST created successfully!\n");
                break;

            case 2:
                if (root == NULL) {
                    printf("\nBST is empty. No traversals to display.\n");
                } else {
                    printf("\n--- BST Traversals ---\n");
                    printf("Inorder  : ");
                    inorder(root);
                    printf("\n");

                    printf("Preorder : ");
                    preorder(root);
                    printf("\n");

                    printf("Postorder : ");
                    postorder(root);
                    printf("\n");
                }
                break;

            case 3:
                if (root == NULL) {
                    printf("\nBST is empty. Cannot search.\n");
                } else {
                    printf("Enter the integer KEY to search: ");
                    scanf("%d", &key);
                    struct Node *result = search(root, key);
                    if (result != NULL) {
                        printf("Success! Element %d found in the BST.\n", key);
                    } else {
                        printf("Result: Element %d NOT found in the BST.\n", key);
                    }
                }
                break;
        }
    } while (choice != 4);
}
```

```

    }
    }
    break;

    case 4:
        printf("Exiting program...\n");
        break;

    default:
        printf("Invalid choice! Please try again.\n");
    }
} while (choice != 4);

return 0;
}

// a. Create a BST by inserting elements recursively
struct Node* insert(struct Node *root, int value) {
    if (root == NULL) {
        struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
        if (!newNode) {
            printf("Memory allocation failed!\n");
            exit(1);
        }
        newNode->data = value;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    // Duplicate values are ignored in standard BST insertion

    return root;
}

// b. Inorder Traversal (Left, Root, Right)
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// b. Preorder Traversal (Root, Left, Right)
void preorder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

```

```
// b. Postorder Traversal (Left, Right, Root)
void postorder(struct Node *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// c. Search the BST for a given element (KEY)
struct Node* search(struct Node *root, int key) {
    // Base cases: root is null or key is present at root
    if (root == NULL || root->data == key) {
        return root;
    }

    // Key is smaller than root's data -> search in left subtree
    if (key < root->data) {
        return search(root->left, key);
    }

    // Key is greater than root's data -> search in right subtree
    return search(root->right, key);
}
```

12. Develop a Program in C for an Employee Record Management System using Hashing with the following operations:

- a. Create a hash table to store employee records using Employee ID as the key.**
- b. Insert employee records into the hash table using the Hash function $H: K \rightarrow L$ as $H(K) = K \bmod m$**
- c. Resolve collisions using Linear Probing.**
- d. Search for an employee record using a given Employee ID and display the details.**
- e. Display the contents of the hash table along with the corresponding memory locations.**

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

#define TABLE_SIZE 10 // Hash table size (m)

// Structure to represent an Employee Record
struct Employee {
    int emp_id; // Key (K)
    char name[50];
    char designation[30];
    float salary;
    int status; // 0: Empty, 1: Occupied, -1: Deleted (optional, using 0 and 1 for simplicity)
};

// Hash Table array initialization
struct Employee hashTable[TABLE_SIZE];

// Function declarations
int hashFunction(int key);
void insertRecord();
void searchRecord();
void displayTable();
```

```

int main() {
    int choice;

    // Initialize hash table slots as empty
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i].status = 0;
    }

    do {
        printf("\n--- Employee Record Management (Hashing with Linear Probing) ---\n");
        printf("1. Insert Employee Record\n");
        printf("2. Search Employee Record by ID\n");
        printf("3. Display Hash Table Contents\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insertRecord();
                break;
            case 2:
                searchRecord();
                break;
            case 3:
                displayTable();
                break;
            case 4:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}

// b. Hash function  $H(K) = K \bmod m$ 
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// a, b, c. Insert employee records using hash function and linear probing
void insertRecord() {
    struct Employee newEmp;
    printf("Enter Employee ID: ");
    scanf("%d", &newEmp.emp_id);
    printf("Enter Name: ");
    scanf(" %[^\\n]s", newEmp.name);
    printf("Enter Designation: ");
    scanf(" %[^\\n]s", newEmp.designation);
    printf("Enter Salary: ");
    scanf("%f", &newEmp.salary);
    newEmp.status = 1;

    int index = hashFunction(newEmp.emp_id);

```

```

int originalIndex = index;
int i = 0;

// c. Resolve collisions using Linear Probing
while (hashTable[index].status == 1) {
    // Handle duplicate keys if needed
    if (hashTable[index].emp_id == newEmp.emp_id) {
        printf("Error: Employee ID %d already exists in the table!\n", newEmp.emp_id);
        return;
    }

    i++;
    index = (originalIndex + i) % TABLE_SIZE;

    // Check if table is completely full
    if (index == originalIndex) {
        printf("\nHash Table Overflow! Cannot insert record, table is full.\n");
        return;
    }
}

// Insert at the available slot found via linear probing
hashTable[index] = newEmp;
printf("Success! Employee record inserted at memory location (index) %d.\n", index);
}

// d. Search for an employee record using a given Employee ID
void searchRecord() {
    int target_id;
    printf("Enter Employee ID to search: ");
    scanf("%d", &target_id);

    int index = hashFunction(target_id);
    int originalIndex = index;
    int i = 0;

    while (hashTable[index].status != 0) {
        if (hashTable[index].status == 1 && hashTable[index].emp_id == target_id) {
            printf("\nEmployee Record Found!\n");
            printf("Memory Location (Index): %d\n", index);
            printf("Employee ID   : %d\n", hashTable[index].emp_id);
            printf("Name          : %s\n", hashTable[index].name);
            printf("Designation   : %s\n", hashTable[index].designation);
            printf("Salary        : %.2f\n", hashTable[index].salary);
            return;
        }

        i++;
        index = (originalIndex + i) % TABLE_SIZE;

        if (index == originalIndex) {
            break; // Checked whole table
        }
    }

    printf("Result: Employee with ID %d not found in the hash table.\n", target_id);
}

```

// e. Display the contents of the hash table along with memory locations

```
void displayTable() {
    printf("\n--- Hash Table Contents (Size: %d) ---\n", TABLE_SIZE);
    printf("%-10s %-12s %-20s %-18s %-10s\n", "Location", "Emp ID", "Name", "Designation", "Salary");
    printf("-----\n");

    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i].status == 1) {
            printf("%-10d %-12d %-20s %-18s %-10.2f\n",
                i,
                hashTable[i].emp_id,
                hashTable[i].name,
                hashTable[i].designation,
                hashTable[i].salary);
        } else {
            printf("%-10d %-12s %-20s %-18s %-10s\n",
                i,
                "---",
                "EMPTY",
                "---",
                "---");
        }
    }
    printf("-----\n");
}
```