University of Engineering and Management
Institute of Engineering & Management, Salt Lake Campus
Institute of Engineering & Management, New Town Campus
University of Engineering & Management, Jaipur

Subject Name: Data Structures and Algorithm Lab

Subject Code: PCCCS391

2nd Year 3rd Semester
IT, CSE(IoT), CSE(IoTCSBT)

Credit: 1	**Lecture Hour :40**

Prerequisite : Programming Fundamentals, Mathematics, C, C++ Programming,

Dr. Sanchita Ghosh, Prof. Angshuman Ray
Prof. Pulak Baral

Course Objective

Obj 1. To implement fundamental data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables using a programming language, and understand their practical applications.

Obj 2. To analyse and compare the efficiency of various searching, sorting, and traversal algorithms in terms of time and space complexity through hands-on coding.

Obj 3. To develop problem-solving skills by designing and implementing efficient algorithms for real-life and theoretical computational problems.

Obj 4. To reinforce theoretical concepts learned in the core Data Structures and Algorithms course through practical experimentation and debugging experience.

Course Outcomes (COs)

CO 1. Apply appropriate data structures such as arrays, linked lists, stacks, queues, trees, and graphs to solve computational problems efficiently.

CO 2. Implement and analyze algorithms for searching, sorting, and traversal with respect to their time and space complexity.

CO 3. Develop modular and reusable code for real-world problems using algorithmic strategies such as divide and conquer, greedy, and dynamic programming.

CO 4. Demonstrate debugging and testing skills to identify and resolve logical and runtime errors in data structure and algorithm implementations.

CO- PO mapping

COs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	1	2	1	1	-	-	-	-	2	1	-	2	-	1	-
CO2	2	1	2	-	-	-	-	-	2	1	-	2	2	-	-
CO3	-	2	2	-	-	-	-	-	1	1	-	2	2	-	-
CO4	-	-	-	2	-	-	-	-	2	1	-	2	-	-	2

Lab Copy : Hand written with index and page numbering

Evaluation Rubric:

	Individual Marking	**Rounded Up to Final Mark**		**Total Marks**
Each Assignment	Properly Executed and Submitted 4 Marks	Done on or before due date 1 Marks	Total= 22x5=110 Map to 50 $\boldsymbol{marks\ obtained 110\times 50}$	50
Total 5 marks per experiment				
Mid Semester	2 Mid-Sem each of 30	$\boldsymbol{midsem1+midsem22}$		30
Mini Project	One project 20	20		20
100				

Assignment 1:

Basics of Pointers, Structures, Functions, DMA, Passing of variables, CLA and Files

Part 1	1. Declare and Initialize Pointers: 1. 1. Declare an integer variable named "num" and initialize it with a value of 20. 1. 2. Declare a pointer variable named "ptr" of type integer and assign it the address of the variable "num". 2. Access and Modify Values Using Pointers: 2.1 Print the value of "num" and the value pointed to by "ptr". 2.2 Modify the value of "num" using the pointer "ptr" to assign a new value. 2.3 Print the updated value of "num" and the value pointed to by "ptr". 3. Pointer Arithmetic: 3.1. Declare an array of integers named "arr" with 5 elements and initialize it with some values. 3.2. Declare a pointer variable named "arrPtr" and assign it the address of the first element of the array. 3.3. Using pointer arithmetic, access and print the values of all the elements in the array using "arrPtr". 4. Pointer to Pointers: 4.1. Declare two integer variables, "a" and "b", and assign them some values. Declare two integer pointers, "ptr1" and "ptr2", and assign them the addresses of "a" and "b" respectively. 4.2. Declare a pointer to a pointer named "pptr" and assign it the address of "ptr1". 4.3. Using the pointer to pointer, print the values of "a" and "b". 4.4. Modify the value of "b" using the pointer to pointer. 4.5. Print the updated value of "b" and the value of "a" using the pointers "ptr2" and "ptr1" respectively.
Part 2	1. Define a Structure: 1. 1. Declare a structure named "Employee" with the following members: a. "name" (string) to store the employee's name b. "id" (integer) to store the employee's ID number c. "salary" (float) to store the employee's salary 1. 2. Declare a variable "emp1" of type "Employee" and initialize its members with sample data. 2. Accessing Structure Members: 2.1 Print the values of the "name," "id," and "salary" members of "emp1" using dot notation. 2.2 Prompt the user to enter values for the members of "emp1" and store them using dot notation. 2.3 Print the updated values of "emp1" members. 3. Array of Structures: 3.1. Declare an array of "Employee" structures named "employeeList" with a size of 3. 3.2. Prompt the user to enter values for the members of each employee in the "employeeList" array. 3.3. Print the details of each employee using a loop and dot notation. 4. Nested Structures:

	4.1. Define a structure named "Date" with members "day" (integer), "month" (integer), and "year" (integer). 4.2. Modify the "Employee" structure from earlier to include a "joiningDate" member of type "Date". 4.3. Prompt the user to enter values for the "joiningDate" member of "emp1" and print the updated details. 5. Passing Structures to Functions: 5.1. Create a function named "printEmployeeDetails" that takes an "Employee" structure as a parameter and prints its details. 5.2. Call the function "printEmployeeDetails" and pass "emp1" as an argument.
Part 3	1. Basic Function Creation and Calling: 1. 1. Create a function named "greet" that takes no parameters and prints a greeting message, such as "Hello, welcome to the lab!" 1. 2. Call the "greet" function from the main function to display the greeting message. 2. Function with Parameters and Return Value: 2.1 Create a function named "addNumbers" that takes two integers as parameters and returns their sum. 2.2 Prompt the user to enter two numbers and store them in variables. 2.3 Call the "addNumbers" function with the user-entered numbers as arguments and print the sum. 3. Recursive Function: 3.1. Create a recursive function named "factorial" that calculates the factorial of a given positive integer. 3.2. Prompt the user to enter a positive integer and store it in a variable. 3.3. Call the "factorial" function with the user-entered number as an argument and print the result. 4. Function with Arrays: 4.1. Create a function named "findMax" that takes an integer array and its size as parameters. 4.2. Inside the function, find the maximum value in the array and return it. 4.3. Declare an integer array and initialize it with some values. 4.4. Call the "findMax" function with the array and its size as arguments and print the maximum value.
Part 4	1. Dynamic Memory Allocation - Single Variable: 1. 1. Prompt the user to enter an integer value. 1. 2. Allocate memory dynamically to store the entered integer using the appropriate function. 1. 3. Assign the entered value to the dynamically allocated memory. 1. 4. Print the value stored in the dynamically allocated memory. 1. 5. Deallocate the dynamically allocated memory. 2. Dynamic Memory Allocation - Array: 2.1 Prompt the user to enter the size of an integer array. 2.2 Allocate memory dynamically to store the integer array of the specified size. 2.3 Prompt the user to enter values for each element of the dynamically allocated array. 2.4 Print the values stored in the dynamically allocated array. 2.5 Deallocate the dynamically allocated memory. 3. Dynamic Memory Allocation - String: 3.1. Prompt the user to enter a string.

	3.2. Allocate memory dynamically to store the entered string using the appropriate function. 3.3. Copy the entered string to the dynamically allocated memory. 3.4. Print the string stored in the dynamically allocated memory. 3.5. Deallocate the dynamically allocated memory. 4. Dynamic Memory Allocation - Structure: 4.1. Define a structure named "Student" with members "name" (string) and "age" (integer). 4.2. Prompt the user to enter the name and age of a student. 4.3. Allocate memory dynamically to store a "Student" structure. 4.4. Assign the entered values to the dynamically allocated structure. 4.5. Print the details of the student stored in the dynamically allocated structure. 4.6. Deallocate the dynamically allocated memory. 5. Dynamic Memory Allocation - 2D Array: 5.1. Prompt the user to enter the number of rows and columns for a 2D integer array. 5.2. Allocate memory dynamically to store the 2D integer array of the specified size. 5.3. Prompt the user to enter values for each element of the dynamically allocated 2D array. 5.4. Print the values stored in the dynamically allocated 2D array. 5.5. Deallocate the dynamically allocated memory.
Part 5	1. Pass by Value vs. Pass by Address: 1. 1. Create a function named "changeValue" that takes an integer parameter. 1. 2. Inside the function, change the value of the parameter to a new value. 1. 3. Call the "changeValue" function from the main function and pass an integer variable as an argument. 1. 4. Print the value of the variable before and after the function call to observe the effect of pass by value. 2. Pass by Value vs. Pass by Address with Arrays: 2.1 Create a function named "modifyArray" that takes an integer array parameter and its size. 2.2 Inside the function, modify the values of the array elements by adding 1 to each element. 2.3 Call the "modifyArray" function from the main function and pass an integer array as an argument along with its size. 2.4 Print the values of the array before and after the function call to observe the effect of pass by value. 3. Pass by Address and Dynamic Memory Allocation: 3.1. Create a function named "doubleValue" that takes an integer pointer parameter. 3.2. Inside the function, double the value of the integer by dereferencing the pointer. 3.3. Allocate memory dynamically for an integer variable in the main function. 3.4. Call the "doubleValue" function from the main function and pass the address of the dynamically allocated integer. 3.5. Print the value of the integer before and after the function call to observe the effect of pass by address. 4. Pass by Value and Structures: 4.1. Create a structure named "Person" with members "name" (string) and

"age" (integer).

4.2. Create a function named "changePersonAge" that takes a "Person" structure as a parameter and modifies its "age" member.

4.3. Call the "changePersonAge" function from the main function and pass a "Person" structure as an argument.

4.4. Print the values of the "age" member before and after the function call to observe the effect of pass by value.

Part 6

1. Understanding Command-Line Arguments Command-line arguments allow you to pass inputs to your C program when you run it from the command line. These inputs can be used to modify the behavior or provide additional information to your program. Arguments are passed as strings separated by spaces.
2. Main Function Declaration In your C program, the main function is where the execution begins. To accept command-line arguments, you need to modify the main function declaration to include two parameters: argc and argv. The argc parameter represents the number of arguments passed, and argv is an array of strings that stores the actual arguments.
   Here's the modified declaration
   of the main function:
   ```
   int main(int argc, char *argv[]) {
               // Code goes here return 0; }
   ```
3. Accessing Command-Line Arguments once you have the main function set up to accept command-line arguments, you can access them within your program using the argv array. The first argument (argv[0]) is always the name of the program itself, and subsequent arguments follow.
   Here's an example that prints all the
   command-line arguments:
   ```
    #include <stdio.h>
   int main(int argc, char *argv[]) {
   //Print all command-line arguments
   for (int i = 0; i < argc; i++) {
   printf("Argument %d: %s\n", i, argv[i]);
   }
   return 0;
   }
   ```
4. Compiling and Running the Program Save the C program with a .c extension (e.g., program.c), and compile it using a C compiler. For example, if you're using gcc, open a terminal and navigate to the directory containing the program. Then, run the following command to compile the program:
   ```
   gcc program.c -o program
   ```
   This command compiles the program and generates an executable file called program. To run the program with command-line arguments, type the following command in the terminal:
   ```
   ./program arg1 arg2 arg3
   ```
   Replace program with the name of your compiled executable and arg1, arg2, arg3, etc., with the arguments you want to pass.
5. Testing the Program Once you execute the program with command-line arguments, it will print each argument along with its index. For example, if you run the program with the command:
   ```
   ./program hello world The output will be:
    Argument 0: ./program
   ```

	Argument 1: hello Argument 2: world You can modify the program to perform different actions based on the passed arguments, depending on your specific requirements.

Part 7	1. File Opening Modes in C: When opening a file in C, you can specify different modes that define how the file will be accessed. Here are some commonly used file opening modes:
	a. - `"r"`: Read mode. Opens an existing file for reading. The file must exist; otherwise, the operation fails.
	b. - `"w"`: Write mode. Opens a file for writing. If the file already exists, its contents are truncated (deleted). If the file does not exist, a new file is created.
	c. - `"a"`: Append mode. Opens a file for appending. If the file exists, new data is written at the end of the file. If the file does not exist, a new file is created.
	2. File Opening, Closing, Reading, and Writing:
	a. Opening a file: To open a file, you can use the `fopen()` function, which returns a `FILE` pointer that can be used to access the file.
	FILE *file = fopen("filename.txt", "mode"); if (file == NULL) {
	// Error handling
	}
	b. Closing a file: After you're done working with a file, it's important to close it using the `fclose()` function to release the resources associated with the file.
	fclose(file);
	c. Reading from a file: You can read data from a file using functions like `fgetc()`, `fgets()`, or `fread()`. These functions allow you to read characters, lines, or binary data, respectively. int ch;
	while ((ch = fgetc(file)) != EOF) {
	// Process the character
	}
	d. Writing to a file: You can write data to a file using functions like `fputc()`, `fputs()`, or `fwrite()`. These functions allow you to write characters, strings, or binary data, respectively.
	int ch = 'A'; fputc(ch, file);
	char *str = "Hello, World!";
	fputs(str, file);
	Note that when using the `"w"` or `"a"` mode, you need to check if the file was opened successfully before writing to it.
	That's a brief overview of file opening, closing, reading, and writing in C. You can use these concepts to perform various file operations in your programs.

Part 8	1. Swap Function Using Pointers: Implement a function that takes two integer pointers as parameters and swaps the values of the integers they point to. Test the function by swapping the values of two variables. 2. Pointer Arithmetic with Structures: Declare a structure named "Student" with members "name" (string) and "age" (integer). Create an array of structures, allocate memory dynamically, and use pointer arithmetic to access and modify the values of the structure members.

3. Pointers and Functions: Create a function that takes an integer array as a parameter and returns a pointer to the maximum element in the array. Test the function by passing an array and printing the maximum value using the returned pointer.
4. Function with Strings: Create a function named "countVowels" that takes a string as a parameter and returns the number of vowels in the string. Test the function by passing different strings and printing the vowel count.
5. Recursive Function with Strings: Create a recursive function named "reverseString" that takes a string as a parameter and reverses it. Test the function by passing different strings and printing the reversed strings.
6. Dynamic Memory Allocation - Structure Array: Create a structure named "Book" with members "title" (string) and "author" (string). Prompt the user to enter the number of books and allocate memory dynamically to store an array of "Book" structures. Prompt the user to enter details for each book and print them.
7. Dynamic Memory Allocation - Resize Array: Create a function that takes an integer array and its current size as parameters. Inside the function, reallocate memory to resize the array to double its size. Test the function by resizing an array and printing its contents.
8. Dynamic Memory Allocation - Matrix Operations: Implement functions to perform matrix addition, multiplication, and transpose operations using dynamically allocated memory for matrices. Test the functions by performing matrix operations on user-entered matrices.
9. Write a C program that acts as a simple calculator. The program should accept three command-line arguments: two numbers and an operator (+, -, *, /). The program should perform the specified arithmetic operation on the given numbers and display the result.
   For example, if the program is called "calculator" and you run it with the command:
   ```
   ./calculator 5 2 +
   ```
   The program should output:
   ```
   5 + 2 = 7
   ```
10. Write a C program that copies the contents of one file to another. The program should accept two command-line arguments: the name of the source file and the name of the destination file. It should read the contents of the source file and write them to the destination file.
    For example, if the program is called "filecopy" and you run it with the command: ./filecopy source.txt destination.txt

The program should copy the contents of "source.txt" to "destination.txt".

Assignment no. 2

Title: Linked List (*Part 1 and Part 2 are mandatory. Part 3 is bonus.*)

Part 1

Create a new C file and name it "linked_list.c".

1. Define a structure called "Node" with the following members:
   i. An integer variable called "data" to store the data of the node.
   ii. A pointer to the next node called "next" to link the nodes together.
2. Declare a global pointer variable called "head" to keep track of the head of the linked list. Initialize it to NULL.
3. Implement the following functions:
   i. void insertNode(int value): This function should insert a new node with the given value at the end of the linked list.
   ii. void deleteNode(int value): This function should delete the node with the given value from the linked list.
   iii. void displayList(): This function should display the elements of the linked list.
   iv. void reverseList(): This function should reverse the order of the nodes in the linked list.
   v. int searchNode(int value): This function should search for a node with the given value in the linked list and return its position (index) if found, or -1 if not found.
4. In the main() function, create a menu-driven program to interact with the linked list. The menu should provide the following options:
   i. Insert a node at the end of the list
   ii. Delete a node by value
   iii. Display the list
   iv. Reverse the list
   v. Search for a node
   vi. Exit the program
5. Ensure that your code follows the standards coding practices, which typically include proper indentation, meaningful variable names, comments, and code modularity.
6. Test your program with various inputs and validate the correctness of each operation.
7. Document your code and include appropriate comments to explain the purpose of each function and significant sections of code.
8. Optimize your code for efficiency and handle edge cases such as an empty list or invalid input.
9. *Submit your code along with a comprehensive report documenting your implementation, including any challenges faced, how you addressed them, and an analysis of the time and space complexity of the different operations.*
10. **[Bonus]**: Move the global pointer named head (point 1b) inside the main function. Now change the function prototypes (point 1c) so that it can be passed as a parameter and works the same. **Note: In practice, usage of global variables is a bad idea.**

Part 2

1. Create a new C file and name it "music_playlist.c".
   a. Define a structure called "Song" with the following members:
      - A character array called "title" to store the song title.
      - A character array called "artist" to store the artist name.
      - A pointer to the next song called "next" to link the songs together.

   b. Declare a pointer (preferably not global) variable called "head" to keep track of the head of the linked list. Initialize it to NULL.

   c. Implement the following functions for music playlist management:
      - void addSong(char title[], char artist[]): This function should add a new song to the linked list.
      - void deleteSong(char title[]): This function should delete the song with the given title from the linked list.
      - void displayPlaylist(): This function should display the details of all the songs in the linked list.
      - void playPlaylist(): This function should simulate playing the songs in the playlist sequentially.
      - int searchSong(char title[]): This function should search for a song with the given title in the linked list and return its position (index) if found, or -1 if not found.

   d. In the main() function, create a menu-driven program to interact with the music playlist. The menu should provide the following options:
      - Add a song to the playlist
      - Delete a song from the playlist
      - Display the playlist
      - Play the playlist
      - Search for a song
      - Exit the program

   e. Implement additional functions as needed to handle the menu options and perform the required operations on the linked list.

   f. Test your program with various inputs to validate the correctness of each operation.

   g. Document your code and include appropriate comments to explain the purpose of each function and significant sections of code.

   h. Optimize your code for efficiency and handle edge cases such as an empty list or invalid input.

   i. Submit your code along with a comprehensive report documenting your implementation, including any challenges faced, how you addressed them, and an analysis of the time and space complexity of the different operations.

   j. Note: You can add additional features to the music playlist application, such as shuffling the playlist, sorting the songs based on artist or duration, or adding a feature to skip songs, to enhance the assignment based on your requirements.

2. Create a new C file and name it "task_manager.c".
   a. Define a structure called "Task" with the following members:
      - An integer variable called "taskId" to store the task ID.
      - A character array called "description" to store the task description.
      - An integer variable called "priority" to store the priority of the task.
      - A pointer to the next task called "next" to link the tasks together.

   b. Declare a pointer (preferably not global) variable called "head" to keep track of the head of the linked list. Initialize it to NULL.

   c. Implement the following functions for task management:

      - void addTask(int taskId, char description[], int priority): This function should add a new task to the linked list.
      - void deleteTask(int taskId): This function should delete the task with the given task ID from the linked list.

- void displayTasks(): This function should display the details of all the tasks in the linked list.
- void prioritizeTasks(): This function should prioritize the tasks based on their priority.
- int searchTask(int taskId): This function should search for a task with the given task ID in the linked list and return its position (index) if found, or -1 if not found.

d. In the main() function, create a menu-driven program to interact with the task manager. The menu should provide the following options:
   - Add a task
   - Delete a task
   - Display all tasks
   - Prioritize tasks
   - Search for a task
   - Exit the program

e. Implement additional functions as needed to handle the menu options and perform the required operations on the linked list.

f. Test your program with various inputs to validate the correctness of each operation.

g. Document your code and include appropriate comments to explain the purpose of each function and significant sections of code.

h. Optimize your code for efficiency and handle edge cases such as an empty list or invalid input.

i. Submit your code along with a comprehensive report documenting your implementation, including any challenges faced, how you addressed them, and an analysis of the time and space complexity of the different operations.

j. Note: You can add additional features to the task manager application, such as sorting the tasks based on priority or due dates, to enhance the assignment based on your requirements.

3. Create a new C file and name it "student_record.c".

a. Define a structure called "Student" with the following members:
   - A character array called "name" to store the student's name.
   - An integer variable called "rollNumber" to store the student's roll number.
   - A floating-point variable called "marks" to store the student's marks.
   - A pointer to the next student record called "next" to link the records together.

b. Declare a global pointer variable called "head" to keep track of the head of the linked list. Initialize it to NULL.

c. Implement the following functions for student record management:
   - void addStudent(char name[], int rollNumber, float marks): This function should add a new student record to the linked list.
   - void deleteStudent(int rollNumber): This function should delete the student record with the given roll number from the linked list.
   - void displayStudents(): This function should display the details of all the students in the linked list.
   - int searchStudent(int rollNumber): This function should search for a student record with the given roll number in the linked list and return its position (index) if found, or -1 if not found.

d. In the main() function, create a menu-driven program to interact with the student record management system. The menu should provide the following options:
   - Add a student record
   - Delete a student record
   - Display all student records
   - Search for a student record

- Exit the program

e. Implement additional functions as needed to handle the menu options and perform the required operations on the linked list.

f. Test your program with various inputs to validate the correctness of each operation.

g. Document your code and include appropriate comments to explain the purpose of each function and significant sections of code.

h. Optimize your code for efficiency and handle edge cases such as an empty list or invalid input.

i. Submit your code along with a comprehensive report documenting your implementation, including any challenges faced, how you addressed them, and an analysis of the time and space complexity of the different operations.

j. Note: You can add additional features to the student record management system, such as sorting the records based on roll numbers or calculating average marks, to enhance the assignment based on your requirements.

Part 3

1. Implement a linked list data structure. Include methods to insert a node at the beginning, at the end, and at a specific position in the list. Test your implementation by creating a linked list and performing various operations on it.
2. Write a function to reverse a linked list in-place. Provide both iterative and recursive implementations for this task.
3. Implement a function to find the middle node of a linked list. If the list has an even number of nodes, return the second middle node.
4. Given two sorted linked lists, write a function to merge them into a single sorted linked list. Preserve the original order of the nodes in each list.
5. Write a function to detect if a linked list contains a cycle. If it does, determine the node at which the cycle starts.
6. Implement a function to remove duplicates from a sorted linked list. The resulting list should only contain unique elements.
7. Implement a function to check if a linked list is a palindrome. Consider both a simple implementation and one that uses a stack for efficient comparison.
8. Write a function to remove the nth node from the end of a linked list. Assume n is always valid ($1 \leq n \leq$ list length).

Implement a function to split a linked list into two separate lists, where one list contains all the even- indexed elements and the other contains all the odd-indexed elements.

Assignment no. 3

Title: Stack (Part 1 and Part 2 are mandatory. Part 3 is bonus.)

Part 1	1. Stack Implementation: a. Define a constant "MAX_SIZE" to represent the maximum size of the stack. b. Define an integer array named "stack" with a size of "MAX_SIZE" to represent the stack. c. Declare an integer variable named "top" and initialize it to -1. This variable will track the index of the topmost element in the stack. d. Create functions for the following stack operations: • "push" - adds an element to the top of the stack. • "pop" - removes the topmost element from the stack. • "isEmpty" - checks if the stack is empty. • "isFull" - checks if the stack is full. • "display" - prints the elements of the stack. e. Implement the functions mentioned above to perform the respective operations. 2. Stack Operations: a. Create an empty stack using the defined stack implementation. b. Push some integer values onto the stack. c. Print the elements of the stack using the "display" function. d. Pop elements from the stack and print the popped elements. e. Check if the stack is empty and print the result. f. Push additional elements onto the stack and print the updated stack.
Part 2	1. Parentheses Matching: a. Implement a function named "isParenthesesMatch" that takes a string as a parameter and checks if the parentheses in the string are balanced. b. The function should use a stack to check for balanced parentheses. c. Test the function by passing strings with balanced and unbalanced parentheses and print the results. 2. Repeat the above assignment (1-2 in Part 1 and 1 in Part 2) using linked lists as a replacement for array.
Part 3	1. Expression Evaluation: Implement a function that takes a mathematical expression in infix notation as a string and evaluates it using the stack-based approach for infix to postfix conversion and postfix evaluation. Test the function by passing different mathematical expressions and printing the results. 2. Stack Reversal: Implement a function that takes a stack as a parameter and reverses its order using an auxiliary stack. Print the elements of the reversed stack to verify the reversal. 3. Stack-based Algorithm: Implement a stack-based algorithm such as postfix expression evaluation, infix to postfix conversion, or infix to prefix conversion. Choose one algorithm and implement it using the stack data structure. Test the function by passing suitable inputs and printing the results. 4. Function Call Stack: The stack is extensively used in programming languages to handle function calls and manage local variables. When a function is called, its execution context, including parameters and local variables, is pushed onto the stack. When the function completes execution, its context is popped from the stack, and control is returned to the calling function. This mechanism allows for nested function calls and efficient memory management. 5. Undo/Redo Operations: Stacks are used to implement the undo/redo functionality in various applications, including text editors, graphic design software, and command-line interfaces. Each performed action is

	stored on a stack, allowing the user to undo the last action by popping it from the stack. The undone actions can be pushed onto a redo stack, enabling the user to redo them if needed. 6. Browser History: Web browsers utilize stacks to maintain the history of visited web pages. Each time a user visits a new page, the URL is pushed onto the stack. When the user clicks the "Back" button, the last visited URL is popped from the stack, allowing navigation to the previous page. Similarly, the "Forward" button can be implemented using a separate stack to store URLs when the user navigates back. 7. Balancing Symbols: Stacks are helpful in checking the balance of symbols, such as parentheses, braces, and brackets, in programming languages. As code is parsed, opening symbols are pushed onto the stack, and when a closing symbol is encountered, it is compared with the topmost symbol on the stack. If they match, the opening symbol is popped, indicating balanced symbols. Any imbalance in the stack indicates an error in the code. 8. Backtracking Algorithms: Various algorithms, like depth-first search (DFS) and backtracking, employ stacks to keep track of the visited nodes or states. In DFS, the stack is used to store unvisited adjacent nodes during traversal, allowing for backtracking to explore other paths. Similarly, backtracking algorithms store the state of the search space on a stack, enabling exploration of alternative paths when needed.

Assignment no. 4

Title: Use of Recursion (Part 1 and Part 2 are mandatory. Part 3 is bonus.)

Part 1	1. Array Sum: a) Implement a recursive function named "arraySum" that calculates the sum of elements in an integer array. b) Create an integer array with a few elements, and then call the "arraySum" function to calculate and display the sum. 2. String Reversal: a) Implement a recursive function named "reverseString" that reverses a given string. b) Prompt the user to input a string, and then call the "reverseString" function to display the reversed string. 3. Palindrome Check: Write a recursive function to determine if a given string is a palindrome. A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward (ignoring spaces, punctuation, and capitalization). 4. Power Function: Create a recursive function to calculate the value of base raised to the power of exponent, where both base and exponent are positive integers. Make sure it handles all the boundary cases. 5. Tower of Hanoi: Solve the classic Tower of Hanoi problem using recursion. Given three pegs and a stack of n disks of different sizes, move the entire stack from one peg to another with the following rules: Only one disk can be moved at a time, and a larger disk cannot be placed on top of a smaller disk. 6. Permutations: Implement a recursive function to generate all possible permutations of a given string. A permutation of a string is an arrangement of its characters in a different order.
Part 2	1. Calculate the Length: Create a recursive function to find the length (number of nodes) of a linked list. The function should count the number of nodes by recursively calling itself with the next node until it reaches the end. 2. Search for an Element: Implement a recursive function to search for a specific value in the linked list. The function should traverse the list by calling itself with the next node until it finds the target value or reaches the end. 3. Merge Two Sorted Lists: Write a recursive function to merge two sorted linked lists into a single sorted linked list. The function should compare the values of the nodes in both lists and recursively merge them to create a new sorted list. 4. Reverse the Linked List: Implement a recursive function to reverse the linked list. The function should recursively reverse the rest of the list and adjust the next pointers as it traverses back. 5. Find the Middle Node: Write a recursive function to find the middle node of a linked list. The middle node is the one located at approximately the center of the list. You can use two pointers to traverse the list—one moving one step at a time and the other moving two steps at a time. 6. Parentheses Balancing: Implement two mutually recursive functions named "isOpenParenthesis" and "isBalanced" to check whether a given string of parentheses is balanced. a. The "isOpenParenthesis" function should call "isBalanced" to check the substring within the parentheses.

	b. The "isBalanced" function should call "isOpenParenthesis" to check if the opening parenthesis has a matching closing parenthesis. c. Prompt the user to input a string containing parentheses, and then call the "isBalanced" function to determine and display whether the parentheses are balanced.
Part 3	1. Palindrome Check: Implement a recursive function to check if a linked list is a palindrome. A linked list is a palindrome if the sequence of its elements is the same when read forward and backward. 2. Combination Sum: Given a set of candidate numbers and a target sum, write a recursive function to find all unique combinations of candidates that sum up to the target. Each number in the candidate set can be used multiple times. 3. Nested List Sum: Write a recursive function to find the sum of all elements in a nested list of integers. The nested list can contain integers or other nested lists. 4. Subset Sum: Write a recursive function to find if there exists a subset of a given set of integers that adds up to a given target sum. The function should return True if such a subset exists; otherwise, return False. 5. Counting Paths: Given a 2D grid of size m x n, write a recursive function to count all possible unique paths from the top-left corner (0, 0) to the bottom-right corner (m-1, n-1). You can only move right or down in the grid. 6. Maze Solver: Given a maze represented as a 2D array where 0 represents an open cell and 1 represents a blocked cell, write a recursive function to find a path from the start point to the end point, if one exists. You can move in any direction (up, down, left, or right) but not diagonally. 7. Expression Evaluation: Write a recursive function to evaluate simple arithmetic expressions represented as strings. The expressions can contain addition, subtraction, multiplication, and division. 8. Binary Search: Implement a recursive function for binary search in a sorted list. Given a sorted list of integers and a target value, write a function to determine if the target value exists in the list and return its index if found, or -1 if not found. 9. Binary Tree Traversals: Implement recursive functions for three types of binary tree traversals: Pre-order, In-order, and Post-order. Traverse a binary tree and print its elements in the specified order.

Assignment no. 5

Title: Queue (Part 1 and Part 2 are mandatory. Part 3 is bonus.)

Part 1	1. Queue Implementation: a. Define a constant "MAX_SIZE" to represent the maximum size of the queue. b. Define an integer array named "queue" with a size of "MAX_SIZE" to represent the queue. c. Declare two integer variables named "front" and "rear" and initialize them to -1. These variables will track the front and rear of the queue, respectively. d. Create functions for the following queue operations: • "enqueue" - adds an element to the rear of the queue. • "dequeue" - removes an element from the front of the queue. • "isEmpty" - checks if the queue is empty. • "isFull" - checks if the queue is full. • "display" - prints the elements of the queue. e. Implement the functions mentioned above to perform the respective operations. 2. Queue Operations: a. Create an empty queue using the defined queue implementation. b. Enqueue some integer values into the queue. c. Print the elements of the queue using the "display" function. d. Dequeue elements from the queue and print the dequeued elements. e. Check if the queue is empty and print the result. f. Enqueue additional elements into the queue and print the updated queue.
Part 2	1. Circular Queue: Implement a circular queue to optimize memory utilization. Modify the enqueue and dequeue functions to handle the circular behavior correctly. 2. Queue using Linked List: Implement a queue data structure using linked lists instead of an array. Create functions to enqueue, dequeue, and display elements in the linked list-based queue. 3. Priority Queue: Implement a priority queue, where each element has an associated priority (or assume it's value as it's priority). Elements with higher priority are dequeued before elements with lower priority.
Part 3	1. Print job management in a printer queue: In this scenario, we will simulate the management of print jobs in a printer queue using the queue data structure. A printer queue is a typical real-world application of a queue where multiple print jobs are received from various users and need to be processed in the order they arrive. a. Define a structure named "PrintJob" with the following members: • "jobID" (integer) to represent the unique ID of the print job. • "jobName" (string) to store the name of the print job (e.g., document name). • "numPages" (integer) to store the number of pages in the print job. b. Implement the Printer Queue: • Create a queue data structure using the queue implementation provided in the previous lab assignment. • The queue will store "PrintJob" structures. c. Simulate Print Job Arrival: • Prompt the user to enter the details of a print job, including the job ID, job name, and number of pages.

- Create a "PrintJob" structure with the entered details.
- Enqueue the created "PrintJob" structure into the printer queue.

d. Print Job Processing:
- Simulate the printing process by dequeueing print jobs from the printer queue.
- Print the details of the dequeued print job (e.g., job ID, job name, and number of pages).
- Simulate the printing process by introducing a delay to mimic the time it takes to print a job.

e. Continuously Receive Print Jobs:
- Implement a loop that allows users to enter multiple print jobs.
- Each entered print job should be enqueued into the printer queue and processed in the order they are received.

f. Optional: Priority Printing:
- Implement a priority queue for print jobs, where the print jobs with fewer pages have higher priority.
- Modify the print job processing to dequeue the print jobs with the highest priority (i.e., fewer pages) first.

g. Note: To simulate the printing process, you can introduce a delay using the "sleep" function (for POSIX systems) or "Sleep" function (for Windows systems). The delay can be a few seconds to mimic the time it takes to print a job. For simplicity, you can use the standard C library's rand () function to generate a random job ID.

h. Ensure that you thoroughly test your simulation by enqueuing multiple print jobs with different page counts and verifying that they are processed in the correct order.

2. Task scheduling in an operating system: In this scenario, we will simulate task scheduling in an operating system using the queue data structure. Task scheduling is a fundamental component of any operating system, where multiple processes and threads compete for execution time on the CPU. The operating system's task scheduler ensures that tasks are executed efficiently and fairly, considering factors like priority and time slice.

a. Define the Process Structure: a) Define a structure named "Process" with the following members:
- "processID" (integer) to represent the unique ID of the process.
- "processName" (string) to store the name of the process.
- "priority" (integer) to indicate the priority of the process (higher value means higher priority).
- "burstTime" (integer) to store the time required for the process to complete its execution.

b. Implement the Task Scheduler Queue:
- Create a queue data structure using the queue implementation provided in the previous lab assignment.
- The queue will store "Process" structures.

c. Simulate Task Arrival:
- Prompt the user to enter the details of a process, including the process ID, process name, priority, and burst time.
- Create a "Process" structure with the entered details.
- Enqueue the created "Process" structure into the task scheduler queue.

d. Task Execution:
- Simulate the task execution process by dequeueing processes from the task scheduler queue.

	• Print the details of the dequeued process (e.g., process ID, process name, priority, and burst time). • Simulate the execution of the process by introducing a delay to mimic the time it takes to execute a task. e. Prioritizing Tasks: • Modify the task scheduling logic to prioritize processes based on their priority values. • Processes with higher priority should be dequeued and executed before processes with lower priority. f. Optional: Round-Robin Scheduling: • Implement a round-robin scheduling algorithm, where each process is given a fixed time slice for execution before being preempted. • Modify the task scheduler to switch between processes after the time slice expires. g. Note: To simulate the execution of tasks, you can introduce a delay using the "sleep" function (for POSIX systems) or "Sleep" function (for Windows systems). The delay can be a few seconds to mimic the time it takes for a process to execute. h. Ensure that you thoroughly test your simulation by enqueuing multiple processes with different priorities and burst times and verifying that they are executed in the correct order based on their priorities. 3. Select a real-world scenario from around your life where a queue data structure is used and implement it. Examples include: a. Supermarket checkout lanes b. Traffic management at intersections c. Call center waiting queues d. Bank Customer Service Queue

Assignment no. 5

Title: Applications of Linear Lists, Stacks and Queue

Part 1	1. Objective: The objective of this lab assignment is to implement a library catalog management system using nested linked lists in the C programming language. 2. Define the Book Structure: Define a structure named "Book" with the following members: a. "title" (string) to store the title of the book. b. "author" (string) to store the name of the author. c. "publicationYear" (integer) to store the year of publication. 3. Define the Genre Structure: Define a structure named "Genre" with the following members: a. "genreName" (string) to store the name of the genre. b. "books" (linked list) to store the list of books belonging to the genre. 4. Implement the Nested Linked List: a. Create a linked list data structure for "Book" structure to represent the list of books under each genre. b. Each node of the "Book" linked list should contain a "Book" structure and a pointer to the next node. c. Create a linked list data structure for "Genre" structure to represent the list of genres in library catalog. d. Each node of the "Genre" linked list should contain a "Genre" structure and a pointer to the next genre. 5. Basic Operations: Implement functions for the following basic linked list operations for the "Book" linked list: a. "createBookNode" - creates a new book node and returns its address. b. "insertBookAtFront" - inserts a new book at the front of the book linked list under a specific genre. c. "deleteBookFromFront" - deletes the first book from the book linked list under a specific genre. d. "displayBooks" - prints the details of all books under a specific genre. 6. Implement functions for the following basic linked list operations for the "Genre" linked list: a. "createGenreNode" - creates a new genre node and returns its address. b. "insertGenreAtFront" - inserts a new genre at the front of the genre linked list. c. "deleteGenreFromFront" - deletes the first genre from the genre linked list. d. "displayGenres" - prints the names of all genres in the library catalog. 7. Library Catalog Management System: a. Provide a user interface for the library catalog management system, allowing users to: • Add new genres to the library catalog. • Add new books under a specific genre. • Delete a genre from the library catalog (and all books under it). • Delete a book from a specific genre. • Display all genres and books in the library catalog. b. Search and Update: Implement functions for the following additional operations:

	• "searchBookByTitle" - searches for a book by title and returns its details (genre, author, publication year). • "updateBookDetails" - allows the user to update the details (author, publication year) of a book by its title. 8. Testing and Demonstration: a. Test your library catalog management system by adding, deleting, and updating genres and books. b. Demonstrate the functionality of the nested linked list by displaying all genres and books in the library catalog.
Part 2	1. Define the Polynomial Structure: a. Define a structure named "Term" with the following members: • "coeff" (integer) to store the coefficient of the term. • "exp" (integer) to store the exponent of the term. • "next" (pointer to the next Term) to create the linked list. 2. Polynomial Creation: a. Implement a function named "createTerm" to create a new Term node and return its address. b. Implement a function named "insertTerm" to insert a new term (with given coefficient and exponent) into the polynomial linked list. c. Prompt the user to input the number of terms in the polynomial and their coefficients and exponents. Create the linked list accordingly. 3. Polynomial Display: Implement a function named "displayPolynomial" to display the polynomial expression in a readable format (e.g., 3x^2 + 2x + 5). 4. Polynomial Addition: a. Implement a function named "addPolynomials" to add two polynomial linked lists and store the result in a new linked list. b. Display the original polynomials and the result polynomial. 5. Polynomial Multiplication: a. Implement a function named "multiplyPolynomials" to multiply two polynomial linked lists and store the result in a new linked list. b. Display the original polynomials and the result of polynomial multiplication. 6. Polynomial Evaluation: a. Implement a function named "evaluatePolynomial" to evaluate the polynomial for a given value of x. b. Prompt the user to input the value of x, and then calculate and display the result of polynomial evaluation. 7. Testing and Demonstration: a. Test each operation (creation, addition, evaluation, and display) with various polynomial expressions. b. Demonstrate the functionality of the program by showing the results of different test cases. 8. Optional Challenge: Polynomial Multiplication by Scalar: a. Implement a function named "multiplyPolynomialByScalar" to multiply a polynomial linked list by a scalar (constant) value. b. Display the original polynomial and the result of polynomial multiplication by scalar. 9. Optional Challenge: Polynomial Differentiation: a. Implement a function named "differentiatePolynomial" to calculate the derivative of the polynomial and store it in a new linked list. b. Display the original polynomial and its derivative. 10. Optional Challenge: Polynomial Integration: a. Implement a function named "integratePolynomial" to calculate the indefinite integral of the polynomial and store it in a new linked list. b. Display the original polynomial and its integral.

Part 3	1. Objective: The objective of this lab assignment is to implement the evaluation of postfix expressions using a stack data structure in the C programming language. 2. Define the Stack Structure: a. Define a structure named "StackNode" with the following members: • "data" (integer) to store the value of the node. • "next" (pointer to the next StackNode) to create the stack. 3. Stack Operations: a. Implement stack operations as necessary, e.g., push, pop, peek etc. 4. Postfix Expression Evaluation: a. Implement a function named "evaluatePostfixExpression" to evaluate a given postfix expression and return the result. b. Prompt the user to input a postfix expression containing numbers and operators (+, -, *, /). c. Implement the algorithm to iterate through the expression, push operands onto the stack, and perform operations when encountering operators. 5. Testing and Demonstration: a. Test the postfix expression evaluation with various postfix expressions. b. Demonstrate the functionality of the program by showing the results of different test cases. 6. Optional Challenge: Handling Expressions with Parentheses: a. Extend the program to handle postfix expressions with parentheses. b. Implement the algorithm to evaluate expressions with parentheses using appropriate stack operations. 7. Extend to Infix to Postfix Conversion: a. Implement a function named "convertInfixToPostfix" to convert an infix expression to a postfix expression. b. Prompt the user to input an infix expression, and then use the stack to convert it to postfix. c. Implement the Shunting Yard algorithm or a similar approach to achieve the conversion. 8. Optional Challenge: Evaluate Infix Expressions: a. Implement a function named "evaluateInfixExpression" to evaluate infix expressions using postfix conversion and evaluation. b. Combine the "convertInfixToPostfix" and "evaluatePostfixExpression" functions to achieve this.

Assignment no. 6

Title: Tree

Part 1	**Task 1** 1. **Binary Tree Creation:** Implement a function to create a binary tree. You can take input in the form of a list or any other suitable data structure. 2. **Traversals:** Implement functions for pre-order, in-order, and post-order traversals of the binary tree. 3. **Height of Binary Tree:** Write a function to find the height (or maximum depth) of the binary tree. 4. **Count Leaf Nodes:** Create a function to count the number of leaf nodes in the binary tree. **Task 2** 1. **BST Insertion:** Implement a function to insert a new node into a Binary Search Tree (BST). 2. **BST Search:** Write a function to search for a given value in the BST. 3. **BST Deletion:** Implement a function to delete a node from the BST. Handle different cases such as a node with no children, one child, and two children. 4. **Find Lowest Common Ancestor:** Write a function to find the Lowest Common Ancestor (LCA) of two nodes in the BST. 5. **Level Order Traversal:** Implement a function to perform level-order traversal (also known as breadth-first traversal) of a binary tree. Print the nodes at each level from left to right. **Task 3** 1. **Check if Binary Tree is Balanced:** Implement a function to check if a binary tree is balanced. A tree is balanced if the height of the left and right subtrees of every node differs by at most one. 2. **Check if Binary Tree is a Binary Search Tree:** Write a function to check if a binary tree is a valid Binary Search Tree. 3. **Diameter of Binary Tree:** Find the diameter (longest path between any two nodes) of the binary tree.
Part 2	**Optional Task 4** 1. **Serialize and Deserialize a Binary Tree:** Implement functions to serialize a binary tree into a string and then deserialize it back into a tree. This is useful for storing and reconstructing binary trees. 2. **Max Path Sum in Binary Tree:** Create a function to find the maximum path sum between any two nodes in a binary tree. The path can start and end at any node in the tree. 3. **Construct Binary Tree from in-order and pre-order Traversals:** Write a function that constructs a binary tree given its in-order and pre-order traversal sequences. 4. **Check if a Binary Tree is a Subtree:** Implement a function to check if a given binary tree is a subtree of another binary tree. 5. **General Tree Creation:** a. Create a general tree where each node can have any number of children. b. Implement traversals for the general tree (e.g., DFS, BFS). c. Create a function to count the number of nodes at level k in the general tree. **Optional Part (Applications)** 1. **File System Implementation:** Create a simplified file system using a tree structure. Nodes represent directories and files. Implement operations like creating files, deleting files, navigating directories, and listing contents.

2. **Binary Expression Tree Evaluation:** Implement a program that builds a binary expression tree from an infix expression and then evaluates it. The nodes of the tree represent operators and operands.
3. **Huffman Coding Compression:** Implement the Huffman coding algorithm using a tree data structure. Build a Huffman tree based on character frequencies and use it to compress and decompress text.
4. **Genealogy/Family Tree:** Design a program that allows users to create and manipulate a family tree. Nodes represent family members, and edges represent parent-child relationships. Implement operations like adding new members, finding ancestors, and listing descendants.
5. **Hierarchical Data Representation:** Use a tree to represent hierarchical data such as organizational structures, family trees, or product category hierarchies. Implement operations like adding new nodes, finding parent/child nodes, and traversing the hierarchy.
6. **Disjoint Set Data Structure (Union-Find):** Implement the disjoint set data structure using a forest of trees. Provide operations for merging sets and finding the representative element of a set.
7. **Spell Checker Using Trie:** Build a spell checker using a trie data structure to efficiently store and search for words. Implement functions for adding words, checking if a word is valid, and suggesting corrections.
8. **Multiway Search Trees (B-trees):** Design and implement a B-tree data structure. Include operations like insertion, deletion, and searching. Test the efficiency of B-trees for large datasets.
9. **Balanced Search Trees (AVL Trees):** Implement an AVL tree, a self-balancing binary search tree. Include operations like insertion, deletion, and searching. Test the efficiency of AVL trees for various operations.
10. **Merkle Trees for Data Integrity:** Build a Merkle tree to ensure data integrity in a distributed system. Implement functions for creating the tree, verifying data, and handling updates.
11. **XML/HTML Parsing Using DOM Tree:** Develop a program to parse XML or HTML documents using a Document Object Model (DOM) tree. Implement operations to traverse and manipulate the tree.
12. **Database Indexing with B+ Trees:** Simulate a database indexing system using B+ trees. Implement operations for adding, deleting, and searching for records in the database.

Assignment no. 7

Title: Searching and Sorting

Part 1	**Part 1** 1. Implement various sorting algorithms e.g., bubble sort, optimized bubble sort, selection sort, insertion sort, merge sort, quick sort and heap sort. 2. Sample inputs of different types and sizes are added in a zip file available in the link. Sample file reading code is also given in the zip. 3. Compare their performance on different input types (sorted, random, almost sorted). 4. Compare their performance on different input sizes (10, 100, 1000, 10000). 5. Analyze the above performances (time taken, number of comparisons, no of swaps, space required, etc.) and present the results visually (e.g., tables, plot as graphs) in the lab report. See this link for reference.
Part 2	**Part 2** 1. Implement linear search using both iterative and recursive approach. 2. Implement binary search using both iterative and recursive approach. 3. (Optional) Implement interpolation search.
Part 3	**Part 3 (Optional)** 1. **Student Grade Sorting:** Write a program that sorts a list of student records based on their grades. Allow users to choose whether to sort by ascending or descending grades. 2. **Online Marketplace Product Listings:** Design a system for sorting and displaying products on an online marketplace. Use sorting algorithms to arrange products based on factors like popularity, price, and ratings. 3. **Top-N Elements:** Implement a program that finds the top N elements from a large dataset. Use a suitable sorting algorithm to efficiently retrieve the highest elements. 4. **Phonebook Application:** Create a phonebook application that allows users to add, delete, and search for contacts. Use an appropriate data structure (e.g., hash table or balanced search tree) for efficient operations.

Assignment no. 8

Title: Graph

Part 1	1. **Graph Representation:** Implement a program to represent a graph using adjacency matrix and adjacency list representations. Provide operations for adding edges and vertices. 2. Create the following (using any graph representation of your choice): a. N (number of nodes in graph): Take input from user (should be > 100). b. Add n vertices to the graph numbered from 1 to N. c. Add an edge between two vertices x and y if x is divisible by y, for all 1<=x,y<=N. d. Find the degrees of all vertices. e. For any given vertex print all its neighbors. f. For any given vertex print all the vertices that are neighbors of its neighbors (but not its direct neighbors). g. For any two/three given vertices print all their common neighbors. h. Find the diameter of the graph. i. Find all prime numbered vertices.
Part 2	1. **Depth-First Search (DFS):** Write a program to perform depth-first search on a graph. Implement both recursive and iterative versions. Print the order in which nodes are visited. 2. **Breadth-First Search (BFS):** Create a program to perform breadth-first search on a graph. Print the order in which nodes are visited. 3. **Shortest Path (Dijkstra's Algorithm):** Implement Dijkstra's algorithm to find the shortest path from a source node to all other nodes in a weighted graph. Print the shortest distances.
Part 3	1. **Minimum Spanning Tree (Prim's Algorithm):** Write a program to find the minimum spanning tree of a connected, weighted graph using Prim's algorithm. Print the edges of the MST. 2. **Minimum Spanning Tree (Kruskal's Algorithm):** Implement Kruskal's algorithm to find the minimum spanning tree of a connected, weighted graph. Print the edges of the MST. 3. **Flight Route Planner:** Create a program that helps users plan flights between cities. Use a graph to represent routes and find the shortest path between two cities. 4. **Social Network Analysis:** Design a program that analyzes a social network graph. Implement functions to find friends of friends, identify cliques, and detect influential nodes. 5. **Detect Cycle (Undirected Graph):** Write a program to detect cycles in an undirected graph. Print whether a cycle is detected or not. 6. **Detect Cycle (Directed Graph):** Implement a program to detect cycles in a directed graph. Print whether a cycle is detected or not.

Mini Project List: Real-World Problems Using DSA

1. **Smart Contact Book with Search and Grouping**
   - **Concepts Used**: Hash tables, Trie, Linked List
   - **Description**: Create a searchable and groupable contact book that supports name-prefix search, grouping by company, and fast retrieval.
2. **Parking Lot Management System**
   - **Concepts Used**: Stacks, Queues, Hash Maps
   - **Description**: Simulate real-time entry and exit of vehicles with efficient space allocation and ticket tracking.
3. **Hospital Patient Record Management System**
   - **Concepts Used**: Priority Queue, Linked List, Trees
   - **Description**: Manage patient details, prioritizing critical patients using min/max heaps and organizing records efficiently.
4. **Autocomplete & Spell Checker**
   - **Concepts Used**: Trie, Hash Maps
   - **Description**: Implement a text-based application that provides real-time word suggestions and correction, similar to search engines.
5. **Movie Recommendation System**
   - **Concepts Used**: Graphs, Hash Maps, Arrays
   - **Description**: Suggest movies based on user history, genre similarity, or friend network using graph traversal and matrix operations.
6. **Train Reservation System**
   - **Concepts Used**: Arrays, Linked Lists, Queues
   - **Description**: Simulate ticket booking, waiting list, and seat assignment using FIFO queue structure and linked list mapping.
7. **Real-Time Stock Span Problem Visualizer**
   - **Concepts Used**: Stack, Arrays
   - **Description**: Track historical stock price patterns and compute span values using stack-based methods.
8. **E-commerce Product Ranking System**
   - **Concepts Used**: Heaps, Priority Queues
   - **Description**: Implement a system that ranks products based on reviews, ratings, and sales using max heaps.
9. **Plagiarism Detection Tool**
   - **Concepts Used**: Hashing, Suffix Trees, KMP Algorithm
   - **Description**: Detect similar content in two documents using efficient string matching and pattern searching algorithms.
10. **Smart City Traffic Route Planner**
    - **Concepts Used**: Graphs, Dijkstra’s Algorithm, A* Search
    - **Description**: Design a system that computes the shortest or fastest path between two city points based on traffic data.