REVA UNIVERSITY
Bengaluru, India

SCHOOL
OF
COMPUTER SCIENCE
AND
ENGINEERING

DATA STRUCTURES LAB
B22EF0305

Third Semester
AY-2023-24
(Prepared in October-2023)

INDEX

1. Lab Objectives:

The objectives of this course are to

1. Understand the principles and importance of data structures in software development.
2. Implement fundamental data structures, such as arrays, linked lists, stacks, queues, trees, and graphs, in C.
3. Apply appropriate data structures to solve various computational problems.
4. Enhance problem-solving skills through hands-on programming exercises.
5. Collaborate effectively in a programming environment by working on team projects.

2. Lab Outcomes:

On successful completion of this course; student shall be able to:

CO#	Course Outcomes	POs	PSOs
CO1	Apply arrays, pointers, and structures concepts to solve real world problems	1 to 5,9,10,12	1
CO2	Demonstrate use different types of Linked Lists to solve real world problems	1 to 5, 9,10,12	1,2
CO3	Illustrate use apply stacks to solve real world problems.	1 to 5, 9,10,12	1,2
CO4	Demonstrate use queues to solve real world problems.	1 to 5, 9,10,12	1,2
CO5	Develop solutions for real world problems using trees.	1 to 5, 9,10,12	1,2,3
CO6	Apply critical thinking and problem-solving skills to solve complex problems involving data structures.	1 to 6, 9,12	1,2,3

3. Lab Requirements:

The following are the required hardware and software for this lab.

Hardware Requirements: A standard personal computer or laptop with the following minimum specifications:

1. **Processor**: Any modern multi-core processor (e.g., Intel Core i5 or AMD Ryzen series).
2. **RAM**: At least 4 GB of RAM for basic programs; 8 GB or more is recommended for larger data structures and complex algorithms.
3. **Storage**: A few gigabytes of free disk space for the development environment and program files.
4. **Display**: A monitor with a resolution of 1024x768 or higher is recommended for comfortable coding.
5. **Input Devices**: Keyboard and mouse (or other input devices) for coding and interacting with the development environment.

Software Requirements:

1. **Operating System**: You can develop and execute C programs for data structures on various operating systems, including Windows, macOS, and Linux.
2. **Text Editor or Integrated Development Environment (IDE)**:
   - **Text Editor:** You can use a simple text editor like Notepad (Windows), Nano (Linux/macOS), or any code-oriented text editor like Visual Studio Code, Sublime Text, or Atom.

- **IDE:** Consider using a C/C++ integrated development environment like Code::Blocks, Dev-C++, or Eclipse CDT for a more feature-rich coding experience.

3. **C Compiler:**
   - To compile and execute C programs, you need a C compiler. GCC (GNU Compiler Collection) is a popular choice for Linux and macOS.
   - For Windows, you can use MinGW (Minimalist GNU for Windows) with GCC or Visual C++ from Microsoft if you prefer a Microsoft development environment.

4. Guidelines to Students:

- Equipment in the lab for the use of the student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage caused is punishable.
- Students are required to carry their observation / programs book with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- The lab can be used in free time / lunch hours by the students who need to use the systems should get prior permission from the lab in-charge.
- Lab records need to be submitted on or before the date of submission.
- Students are not supposed to use flash drives.

PART – A

LAB EXPERIMENTS AND ASSIGNMENTS

5. List of Lab Experiments and Assignments:

Sl No.	Title of the Experiment
	Part A
1.	**Pointers and Arrays:** You are working on a genetics research project and have been tasked with creating a C program to analyze genetic sequences. Genetic sequences are represented as strings of characters **(A: Adenine, T: Thymine, C: Cytosine,** and **G: Guanine)**, where each character represents a nucleotide in the DNA sequence. You need to develop a C program that performs basic analysis on genetic sequences provided by the user. The program will use dynamic memory allocation to store the genetic sequences and arrays to store the analysis results. The program will have the following functionalities: 1. **Input Genetic Sequences:** The program should prompt the user to input a number of genetic sequences. Each sequence should be stored dynamically in memory. 2. **Analysis Functions**: a. **Count Nucleotides**: Write a function that takes a genetic sequence as input and counts the occurrences of each nucleotide (A, C, G, and T) in that sequence. Store the counts in an array. b. **Calculate GC Content**: Write a function that calculates the GC content (the percentage of G and C nucleotides) of a genetic sequence. Store the GC content in an array. The GC content of a DNA sequence is calculated using the following formula: 3. **Display Results**: After analyzing all the input sequences, display the analysis results for each sequence. Display the counts of each nucleotide and the calculated GC content. **Requirements**: • Use dynamic memory allocation to store genetic sequences. • Use arrays to store the analysis results (nucleotide counts and GC content) for each sequence. • Implement separate functions for counting nucleotides and calculating GC content. • Use appropriate data types and structures to store the information. **Sample Output**: Genetic Sequence Analyzer Enter the number of genetic sequences: 3 Enter genetic sequence 1: ATCGATTAGCGTCA Enter genetic sequence 2: GCGCGCGTGCAT Enter genetic sequence 3: TACGTACGTACGTACG Analysis Results: Sequence 1: A: 3 C: 3 G: 3 T: 3 GC Content: 33.33% Sequence 2: A: 2 C: 5 G: 5 T: 1 GC Content: 70.00% Sequence 3:

	A: 4 C: 4 G: 4 T: 4 GC Content: 50.00% **Note:** In this experiment, you're practicing dynamic memory allocation and array usage. The scenario involves real-world genetics data analysis, making it more engaging and practical.
2.	**Pointers and Structures:** You are tasked with simulating the behavior of particles in a 2D space using C programming. Each particle is represented by its position (`**x**` and `**y**` coordinates as **floats**) and velocity (`**vx**` and `**vy**` components as **floats**). Your goal is to write a program that does the following: 1. Define a structure named `**Particle**` with attributes for position and velocity components. 2. Declare an array of `Particle` structures to store information for a maximum of 4 particles. 3. Populate the array with sample data for 3 particles, including their positions and velocities. 4. Write a function `updateParticle` that takes a pointer to a `Particle` structure as its argument and updates the particle's position based on its velocity. You can assume a constant time step for simplicity. 5. Write a function `printParticle` that takes a pointer to a `Particle` structure as its argument and displays the particle's position and velocity in a formatted manner. 6. In the `main` function, simulate the movement of particles by repeatedly calling the `updateParticle` function for each particle and then calling the `printParticle` function to display their updated information. Provide the C code to implement the above scenario. **Note:** In this experiment, you're practicing pointers and structure usage. The scenario involves particle simulation in physics.
3.	**Dynamic Memory Allocation, Arrays, and Structures: Tic-Tac-Toe Game** In this programming experiment, you will implement a Tic-Tac-Toe game in C that utilizes dynamic memory allocation, arrays, and structures. The game will be played between two players, **'X'** and **'O'**, on a 3x3 grid. The program will dynamically allocate memory for the grid and use structures to represent the players and their moves. 1. Create a structure named `Player` with the following attributes: a. **`char symbol:`** To store the player's symbol (**'X'** or **'O'**). b. **`int score`**: To store the player's score (**1** for win, **0** for loss). 2. Implement a function named **`initializeGrid`** that dynamically allocates memory for a 3x3 grid. Initialize the grid with empty spaces (' '). 3. Implement a function named **`printGrid`** that takes the dynamically allocated grid as input and prints the current state of the Tic-Tac-Toe grid. 4. Implement a function named **`checkWin`** that checks if a player has won the game. It should take the grid and the current player's symbol as input and return 1 if the player has won, 0 otherwise. 5. Implement the main function: a. Initialize two instances of the `Player` structure, one for **'X'** and one for **'O'**.

b. Use a loop to alternate between players and allow them to make moves.

c. For each move, prompt the current player to enter their row and column choices.

d. Validate the inputs to ensure they are within the valid range and the chosen cell is not already occupied.

e. After each move, update the grid and print its current state.

f. Check if the current player has won using the **`checkWin`** function.

g. If a player wins, update their score and display the winner.

h. If the grid is full and no one has won, display a tie message.

6. Properly deallocate the dynamically allocated memory for the grid at the end of the program.

7. Use proper error handling and input validation to handle incorrect user inputs gracefully.

Note: This exercise focuses on dynamic memory allocation, arrays and structures.

4. **Doubly Linked List:**

A linked list is a linear data structure used in computer science and programming for storing and organizing a collection of elements. A linked list can be visualized as a chain of nodes, where each node points to the next node, creating a sequence. Create a C program on a doubly linked list that has a header node which maintains additional information of the list like – Number of nodes. Perform insertion at beginning, at end, at a position in the list, perform deletion at beginning, at end and of an item from the list, also include a display operation to display the contents of the list. Your program must be a menu driven program that provides choices to perform various operations. The node should contain the SRN, Name, branch and semester information of students in a college.

Note: In this exercise, you will practice implementing a doubly linked list.

5. **Implementing a Stack using Linked List:**

You are required to implement all the basic stack operations i.e., push, pop, peek, isEmpty, and display.

The following are the rubrics to be followed while preparing the code.

1. **Data Structures**:
   a. Define a structure **Node** to represent each element of the linked list. It should contain two fields: a data field to store the value, and a pointer to the next node.
   b. Define a structure **Stack** to represent the stack. It should contain a pointer to the top node of the linked list.
2. **Initialization**:
   a. Implement a function **void initialize(Stack *stack)** to initialize the stack. This function should set the top pointer to **NULL**.
3. **Push Operation**:
   a. Implement a function **void push(Stack *stack, int value)** to push an element onto the stack.
   b. Create a new node with the provided value and link it to the top of the stack.
4. **Pop Operation**:
   a. Implement a function **int pop(Stack *stack)** to pop an element from the stack.
   b. Return the value of the top node, update the top pointer, and free the memory of the popped node.
5. **Peek Operation**:
   a. Implement a function **int peek(Stack *stack)** to retrieve the value of the top element without removing it.
   b. Return the value of the top node without modifying the stack.
6. **isEmpty Operation**:

	a. Implement a function **int isEmpty(Stack *stack)** to check if the stack is empty. b. Return 1 if the stack is empty, and 0 otherwise. 7. **Display Operation**: a. Implement a function **void display(Stack *stack)** to display the elements of the stack from top to bottom. b. Traverse through the linked list and print each element's value. 8. **Testing**: a. Create a menu driven main function to test your stack implementation. b. Perform a sequence of push, pop, and peek operations to demonstrate the functionality of the stack. c. Test edge cases, such as popping from an empty stack and peeking into an empty stack, to ensure proper error handling. Implement error handling for edge cases, such as stack overflow and underflow. For example, if the stack is full and a push operation is attempted, or if a pop operation is attempted on an empty stack, display appropriate error messages. **Note**: In this exercise, you will practice implementing a stack data structure using a linked list.
6.	**Circular Queue Implementation with Linked Storage:** You have been tasked with implementing a circular queue data structure using linked storage in C language. The queue should include a header node that stores the number of nodes in the queue, as well as the largest and smallest values currently present in the queue. Implement the following operations for this circular queue: 1. **Initialize Queue**: Write a function to initialize an empty circular queue with a header node. The header node should have its count initialized to 0, and largest and smallest initialized to appropriate values. 2. **Enqueue**: Write a function to enqueue an integer value into the circular queue. Update the header node's count, largest, and smallest accordingly. 3. **Dequeue**: Write a function to dequeue an element from the circular queue. Update the header node's count, largest, and smallest appropriately. 4. **Get Largest Value**: Write a function to retrieve the largest value from the circular queue. 5. **Get Smallest Value**: Write a function to retrieve the smallest value from the circular queue. 6. **Display Queue**: Write a function to display the elements of the circular queue, starting from the front. **Note**: In this exercise, you will practice implementing a queue data structure using a linked list.
7.	**Infix to Postfix Conversion using Dynamic Stack:** Write a C program to convert an infix expression containing the operators +, -, *, /, %, and ^, along with parentheses, into its postfix form. Your program should use a dynamic flexible array to implement the stack for the conversion process. The program should also be case-insensitive, meaning that it should recognize operators and operands regardless of their case (upper or lower). Your program should perform the following steps: 1. Read an infix expression from the user. 2. Convert the infix expression to postfix using the dynamic stack. 3. Display the resulting postfix expression. Remember to handle operator precedence and associativity correctly during the conversion process.
8.	**Evaluating Postfix Expressions using Linked List-based Stack:**

	Write a C program to evaluate a postfix expression using a linked list to implement the stack. The program should validate inputs to ensure that only numbers and valid operators are accepted. Your program should perform the following steps: 1. Define a structure for a linked list node that holds an integer value and a pointer to the next node. 2. Implement functions to perform stack operations (push, pop, isEmpty, etc.) using the linked list. 3. Implement a function to evaluate a postfix expression. The function should accept a string containing the postfix expression and return the result of the evaluation. 4. The program should validate the input expression to ensure that only numbers and valid operators (+, -, *, /, %) are present. Any other characters should be considered invalid input. 5. Display appropriate error messages if the input expression is invalid or if there's a problem during evaluation.
9.	**Round Robin Scheduling Simulation – Queues:** You have been tasked with implementing a round robin scheduling simulation using a dynamically allocated array-based queue in C. Your goal is to help manage a set of processes and simulate their execution based on the round robin scheduling algorithm, assuming time of arrival is same for all processes. Write a C program that accomplishes the following tasks: 1. Prompt the user to enter the number of processes. 2. For each process, prompt the user to enter the burst time. 3. Prompt the user to enter the time quantum for the round robin scheduling. 4. Implement the round robin scheduling algorithm using a dynamically allocated array-based queue. 5. Display a Gantt chart that shows the execution order of processes along with their respective start and end times. 6. Ensure that the queue is properly managed for task scheduling. 7. Free any dynamically allocated memory before the program exits.
10.	**Complete Binary Tree:** You are tasked with implementing a program that creates a complete binary tree from user input and offers different traversal options. Complete binary trees are special binary trees where all levels are completely filled except possibly the last level, and nodes are added from left to right. Your program should perform the following steps: 1. Take input from the user for the number of elements `n` and then `n` elements to create a complete binary tree. 2. Implement a function to insert elements into the complete binary tree in level order. 3. Implement functions to perform inorder, preorder, and postorder traversals of the created binary tree. 4. Provide a menu-driven main function that allows the user to choose from the following options: a. Inorder Traversal b. Preorder Traversal c. Postorder Traversal d. Exit Your implementation should create the complete binary tree and allow the user to select a traversal method from the menu. Each traversal method should display the corresponding order of node values.

	Write a complete C program to achieve the above functionality. You are free to use any standard C libraries and programming constructs.
11.	**Binary Tree Routing Table** Create a C program to demonstrate the use of binary search tree (BST) in implementing a basic routing table for a network. Binary search trees are a type of binary tree where each node has at most two children, and the values in the left subtree are smaller than the node's value, while the values in the right subtree are greater. This property allows for efficient searching and insertion. The program should output appropriate nextHop or "Route not found" message based on destination IP address. The program should do the following tasks: **Binary Search Tree (BST):** In a binary search tree routes must be organized based on the destination IP addresses. The left subtree of a node contains routes with smaller destination IP addresses, and the right subtree contains routes with larger destination IP addresses. **Route Insertion Criteria:** Create a function insert to insert a new route into the binary search tree. • If the tree is empty (i.e., root is NULL), a new node is created with the route and returned. • If the tree is not empty, the function compares the destination IP address of the new route with the destination IP address of the current node. • If the new route's destination IP is less than the current node's destination IP, the insertion continues in the left subtree recursively. • If the new route's destination IP is greater than or equal to the current node's destination IP, the insertion continues in the right subtree recursively. • The function returns the modified subtree after inserting the new route. This structure ensures that when performing a routing lookup, the program can efficiently traverse the tree to find the appropriate next hop based on the destination IP address. **Look up**: Create lookup function for performing a routing lookup in the binary search tree (BST) to find the appropriate next hop for a given destination IP address. **Delete Node:** Create a function **deleteNode** to delete a node with a specific destination IP address from the binary tree routing table.
12.	**Implementing a Symbol Table using Binary Search Tree (BST)** In this programming exercise, you will have the opportunity to implement a symbol table using a Binary Search Tree (BST). A symbol table is a fundamental data structure used in various applications to store key-value pairs. In this case, you will be creating a symbol table that stores words and their corresponding frequencies. Your task is to implement a C program that demonstrates the use of a Binary Search Tree as a symbol table. The program should allow users to input words, which will be inserted into the BST. Each word's frequency will be tracked. After inserting words, the program should display the contents of the symbol table and prompt the user to enter a word for searching. If the word is found in the symbol table, its frequency should be displayed; otherwise, a message indicating that the word is not in the table should be shown.
	Part B (Projects)
	Students have to do a project assigned to them.

6. Session – 1: Lab Experiment - Pointers and Arrays:

- **Problem Statement:**

You are working on a genetics research project and have been tasked with creating a C program to analyze genetic sequences. Genetic sequences are represented as strings of characters **(A: Adenine, T: Thymine, C: Cytosine,** and **G: Guanine)**, where each character represents a nucleotide in the DNA sequence.

You need to develop a C program that performs basic analysis on genetic sequences provided by the user. The program will use dynamic memory allocation to store the genetic sequences and arrays to store the analysis results. The program will have the following functionalities:

1. **Input Genetic Sequences:** The program should prompt the user to input a number of genetic sequences. Each sequence should be stored dynamically in memory.
2. **Analysis Functions**:
    a. **Count Nucleotides**: Write a function that takes a genetic sequence as input and counts the occurrences of each nucleotide (A, C, G, and T) in that sequence. Store the counts in an array.
    b. **Calculate GC Content**: Write a function that calculates the GC content (the percentage of G and C nucleotides) of a genetic sequence. Store the GC content in an array. The GC content of a DNA sequence is calculated using the following formula:

$$GC\ Content = \frac{Number\ of\ G\ and\ C\ nucleotides}{Total\ number\ of\ nucleotides} \times \mathbf{100}$$

3. **Display Results**: After analyzing all the input sequences, display the analysis results for each sequence. Display the counts of each nucleotide and the calculated GC content.

Requirements:

- Use dynamic memory allocation to store genetic sequences.
- Use arrays to store the analysis results (nucleotide counts and GC content) for each sequence.
- Implement separate functions for counting nucleotides and calculating GC content.
- Use appropriate data types and structures to store the information.

Sample Output:

```
Genetic Sequence Analyzer
Enter the number of genetic sequences: 3
Enter genetic sequence 1: ATCGATTAGCGTCA
Enter genetic sequence 2: GCGCGCGTGCAT
Enter genetic sequence 3: TACGTACGTACGTACG
Analysis Results:
Sequence 1:
A: 3
C: 3
G: 3
T: 3
GC Content: 33.33%
Sequence 2:
A: 2
C: 5
G: 5
T: 1
GC Content: 70.00%
Sequence 3:
A: 4
C: 4
G: 4
T: 4
GC Content: 50.00%
```

Note:

In this experiment, you're practicing dynamic memory allocation and array usage. The scenario involves real-world genetics data analysis, making it more engaging and practical.

- **Student Learning Outcomes**

After successful execution of this program, the student shall be able to

- Use Dynamic Memory Allocation in C to allocate arrays dynamically.
- Apply Control Structures and Functions in C programs.

- **Theory**

Dynamic arrays, also known as resizable arrays, are data structures that provide a flexible way to store and manage collections of elements in computer memory. Unlike static arrays, where the size is fixed at the time of declaration and cannot be changed during runtime, dynamic arrays can grow or shrink as needed.

Dynamic arrays are implemented using dynamic memory allocation techniques, typically through functions like malloc and realloc in C/C++ or ArrayList and Vector classes in languages like Java and C#.

When you create a dynamic array, it starts with a certain initial capacity or size. This capacity can be smaller or equal to the number of elements initially added to the array.

As elements are added to a dynamic array and it reaches its current capacity, the array is resized to accommodate more elements.

Dynamic arrays are commonly used in situations where you need a flexible data structure for managing a collection of items whose size may vary over time. They are often used in implementing lists, stacks, queues, and other data structures.

Dynamic arrays are a fundamental data structure in many programming languages and libraries because they strike a balance between the efficiency of static arrays and the flexibility of linked lists. Understanding how to use them effectively is an important skill for software developers.

- **Algorithm:**

1. Start the program.
2. Declare and initialize necessary variables:
    - `numSequences`: The number of genetic sequences to be analyzed.
    - `sequences`: A dynamic array of character pointers (strings) to store the genetic sequences.
    - `nucleotideCounts`: A dynamic array of integer arrays to store the counts of 'A', 'C', 'G', and 'T' nucleotides for each sequence.
    - `gcContent`: A dynamic array of doubles to store the GC content for each sequence.
3. Display a welcome message: "Genetic Sequence Analyzer."
4. Prompt the user to enter the number of genetic sequences (`numSequences`).
5. Allocate memory for the dynamic arrays:
    - Allocate memory for `sequences` to store `numSequences` pointers to strings.
    - Allocate memory for `nucleotideCounts` to store `numSequences` pointers to integer arrays (each with 4 elements).
    - Allocate memory for `gcContent` to store `numSequences` double values.
6. Use a loop to input genetic sequences and perform analysis for each sequence:
    - For each sequence (index `i` from 0 to `numSequences - 1`):
        - Allocate memory for `sequences[i]` to store the genetic sequence (maximum length assumed to be 100 characters).
        - Allocate memory for `nucleotideCounts[i]` to store counts of 'A', 'C', 'G', and 'T' nucleotides (initialize to zero).
        - Initialize `gcContent[i]` to 0.0.
        - Prompt the user to enter genetic sequence `i + 1`.
        - Read the genetic sequence into `sequences[i]`.
        - Call the `countNucleotides` function to count the nucleotides in the sequence and store the counts in `nucleotideCounts[i]`.

- Call the `calculateGCContent` function to calculate the GC content for the sequence and store it in `gcContent[i]`.

7. Display the analysis results:
   - For each sequence (index `i` from 0 to `numSequences - 1`):
     - Display "Sequence i+1:" to indicate the sequence number.
     - Display the counts of 'A', 'C', 'G', and 'T' nucleotides using `nucleotideCounts[i]`.
     - Display the GC content using `gcContent[i]` with two decimal places.
     - Add a newline for separation.
8. Free the allocated memory to prevent memory leaks:
   - Use a loop to free memory for each genetic sequence in `sequences` and their respective `nucleotideCounts`.
   - Free memory for `sequences`, `nucleotideCounts`, and `gcContent`.
9. End the program.

- **Source Code:**

```
#include <stdio.h>
#include <stdlib.h>
// Function to count nucleotides in a genetic sequence
void countNucleotides(char *sequence, int *counts) {
    for (int i = 0; sequence[i] != '\0'; i++) {
        switch (sequence[i]) {
            case 'A':
                counts[0]++;
                break;
            case 'C':
                counts[1]++;
                break;
            case 'G':
                counts[2]++;
                break;
            case 'T':
                counts[3]++;
                break;
        }
    }
}
// Function to calculate GC content of a genetic sequence
double calculateGCContent(char *sequence) {
    int totalNucleotides = 0;
    int gcCount = 0;
    for (int i = 0; sequence[i] != '\0'; i++) {
        if (sequence[i] == 'G' || sequence[i] == 'C') {
            gcCount++;
        }
        totalNucleotides++;
    }
    return ((double)gcCount / totalNucleotides) * 100.0;
}
// Main function
int main() {
    int numSequences;
    printf("Genetic Sequence Analyzer\n\n");
    printf("Enter the number of genetic sequences: ");
    scanf("%d", &numSequences);
    char **sequences = (char **)malloc(numSequences * sizeof(char *));
    int **nucleotideCounts = (int **)malloc(numSequences * sizeof(int *));
    double *gcContent = (double *)malloc(numSequences * sizeof(double));
    for (int i = 0; i < numSequences; i++) {
        sequences[i] = (char *)malloc(100 * sizeof(char)); // Assuming max sequence length of 100 characters
```

```
        nucleotideCounts[i] = (int *)calloc(4, sizeof(int)); // Initialize counts to zero
        gcContent[i] = 0.0;
        printf("Enter genetic sequence %d: ", i + 1);
        scanf("%s", sequences[i]);
        countNucleotides(sequences[i], nucleotideCounts[i]);
        gcContent[i] = calculateGCContent(sequences[i]);
    }
    printf("\nAnalysis Results:\n");
    for (int i = 0; i < numSequences; i++) {
        printf("Sequence %d:\n", i + 1);
        printf("A: %d\n", nucleotideCounts[i][0]);
        printf("C: %d\n", nucleotideCounts[i][1]);
        printf("G: %d\n", nucleotideCounts[i][2]);
        printf("T: %d\n", nucleotideCounts[i][3]);
        printf("GC Content: %.2f%%\n", gcContent[i]);
        printf("\n");
    }
    // Free allocated memory
    for (int i = 0; i < numSequences; i++) {
        free(sequences[i]);
        free(nucleotideCounts[i]);
    }
    free(sequences);
    free(nucleotideCounts);
    free(gcContent);
    return 0;
}
```

- **Explanation of the program:**

The purpose of "Genetic Sequence Analyzer" is to analyze genetic sequences provided by the user. Here is the explanation of how the code works:

1. The program starts by including two standard C libraries, `<stdio.h>` for input/output operations and `<stdlib.h>` for memory allocation and deallocation.
2. **Function Declarations**: Two functions are declared at the beginning of the code:
    - `countNucleotides`: This function counts the occurrences of each nucleotide (A, C, G, T) in a given genetic sequence and stores the counts in an integer array.
    - `calculateGCContent`: This function calculates the GC content (percentage of G and C nucleotides) of a genetic sequence.
3. **Main Function**: The `main` function is where the program execution begins.
    - It starts by declaring an integer variable `numSequences` to store the number of genetic sequences the user wants to analyze.
    - User Input:
        - The program prompts the user to enter the number of genetic sequences they want to analyze.
        - It allocates memory to store the sequences, nucleotide counts, and GC content for each sequence based on the user's input.
    - Loop for Analyzing Sequences:
        - For each genetic sequence (up to the user-specified number), the program allocates memory for storing the sequence, initializes nucleotide counts to zero, and initializes GC content to 0.0.
        - It then prompts the user to enter a genetic sequence, which is read as a string and stored in the `sequences` array.
        - The `countNucleotides` function is called to count the occurrences of A, C, G, and T in the sequence, and these counts are stored in the `nucleotideCounts` array.
        - The `calculateGCContent` function is called to calculate the GC content of the sequence and store it in the `gcContent` array.
    - Displaying Analysis Results:
        - After analyzing all the user-provided sequences, the program displays the analysis results for each sequence.

        - It shows the counts of A, C, G, and T nucleotides, as well as the GC content as a percentage for each sequence.
    - Memory Cleanup:
        - Finally, the program frees the dynamically allocated memory to prevent memory leaks.
4. **Memory Management**:
    - Memory is allocated for:
        - An array of character pointers (`sequences`) to store genetic sequences.
        - An array of integer arrays (`nucleotideCounts`) to store counts of A, C, G, and T for each sequence.
        - An array of double values (`gcContent`) to store the GC content for each sequence.
    - Memory is freed for all these arrays at the end of the program to release the allocated memory.

Overall, this program allows the user to input genetic sequences, counts the occurrences of each nucleotide, calculates the GC content, and presents the results for multiple sequences. It demonstrates basic memory allocation and deallocation techniques in C.

- **Expected Results**

Genetic Sequence Analyze
Enter the number of genetic sequences: 2
Enter genetic sequence 1:
ATGCGTACGTACGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAG
CTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
Enter genetic sequence 2:
CGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCG
CGCGCGCGCGC

Analysis Results:
Sequence 1:
A: 100
C: 100
G: 200
T: 0
GC Content: 66.67%

Sequence 2:
A: 0
C: 50
G: 50
T: 0
GC Content: 100.00%

7. Session – 2: Lab Experiment - Pointers and Structures:

- **Problem Statement:**

You are tasked with simulating the behavior of particles in a 2D space using C programming. Each particle is represented by its position (`**x**` and `**y**` coordinates as **floats**) and velocity (`**vx**` and `**vy**` components as **floats**).

Your goal is to write a program that does the following:

1. Define a structure named `**Particle**` with attributes for position and velocity components.
2. Declare an array of `**Particle**` structures to store information for a maximum of 4 particles.
3. Populate the array with sample data for 3 particles, including their positions and velocities.
4. Write a function `**updateParticle**` that takes a pointer to a `Particle` structure as its argument and updates the particle's position based on its velocity. You can assume a constant time step for simplicity.
5. Write a function `**printParticle**` that takes a pointer to a `Particle` structure as its argument and displays the particle's position and velocity in a formatted manner.
6. In the `main` function, simulate the movement of particles by repeatedly calling the `**updateParticle**` function for each particle and then calling the `**printParticle**` function to display their updated information.

Provide the C code to implement the above scenario.

Note:

In this experiment, you're practicing pointers and structure usage. The scenario involves particle simulation in physics.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- Create a structure in C and use them in application programs.
- Make use of pointers with structures in creating applications.

- **Theory:**

In C programming, a structure is a user-defined data type that allows you to group together variables of different data types under a single name. Structures provide a way to organize and store related data efficiently.

Definition: A structure is defined using the `struct` keyword, followed by a structure tag (or name), and a list of member variables enclosed in curly braces.

For example:

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

Declaration: To declare a structure variable, you use the defined structure tag, followed by the variable name.

For example:

```
struct Person person1;
```

Structure Members: A structure can have one or more members, which are variables of different data types. In the example above, `name`, `age`, and `height` are members of the `Person` structure.

Accessing Structure Members: To access the members of a structure, you use the dot (`.`) operator or arrow (→) operator through the pointer variable.

Memory Allocation: Structures are allocated contiguous memory in the order in which their members are declared. The size of a structure is the sum of the sizes of its individual members. Padding may be added for alignment purposes to ensure efficient memory access.

Nested Structures: Structures can be nested inside other structures, allowing you to create complex data structures.

Passing Structures to Functions: You can pass structures to functions by value or by reference (using pointers). When passing by value, a copy of the entire structure is made. When passing by reference, changes made to the structure inside the function affect the original structure.

Typedef: The `typedef` keyword is often used to create an alias for a structure, making it more concise to declare structure variables.

For example:

```
typedef struct {
      int x;
      int y;
} Point;
Point p1;
```

Use Cases: Structures are commonly used to represent real-world entities or complex data structures like linked lists, trees, and graphs. They are also essential in file I/O, allowing you to read and write structured data to and from files.

Understanding structures is fundamental in C programming as they enable you to create organized and versatile data structures, facilitating the management of complex data in your programs. Structures provide a way to represent and work with structured data effectively.

- **Algorithm:**

1. Define a structure `Particle` to represent a particle with attributes `x`, `y` (position), `vx`, and `vy` (velocity).
2. Create two functions:
   - `updateParticle(struct Particle *particle)`: This function updates the position of a particle by adding its velocity to its current position.
   - `printParticle(struct Particle *particle)`: This function prints the position and velocity of a particle.
3. In the `main` function:
   - Seed the random number generator using `srand(time(NULL))` to ensure different random values on each program run.
4. Declare an array of `Particle` structures, `particles`, with a size of 4.
5. Generate random particles inside a loop:
   - For `i` from 0 to 2 (3 times):
     - Set `particles[i].x` to a random float value between 0 and 10.
     - Set `particles[i].y` to a random float value between 0 and 10.
     - Set `particles[i].vx` to a random float value between -1 and 1.
     - Set `particles[i].vy` to a random float value between -1 and 1.
6. Iterate through the generated particles:
   - For each particle:
     - Print the particle's index (e.g., "Particle 1:").
     - Print the initial position and velocity using `printParticle`.
     - Enter a loop that simulates 5 steps of particle movement:
       - Inside the loop:
         - Update the particle's position using `updateParticle`.
         - Print the updated position and velocity after each step using `printParticle`.

- **Source Code:**

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

struct Particle {
   float x, y;     // Position
   float vx, vy;   // Velocity
};
void updateParticle(struct Particle *particle) {
   particle->x += particle->vx;
```

```
    particle->y += particle->vy;
}
void printParticle(struct Particle *particle) {
    printf("Position: (%.2f, %.2f)\n", particle->x, particle->y);
    printf("Velocity: (%.2f, %.2f)\n\n", particle->vx, particle->vy);
}
int main() {
    // Seed the random number generator
    srand(time(NULL));
    struct Particle particles[4];
    // Generate random particles
    for (int i = 0; i < 3; i++) {
      particles[i].x = (float)(rand() % 100) / 10.0; // Random position between 0 and 10
      particles[i].y = (float)(rand() % 100) / 10.0; // Random position between 0 and 10
      particles[i].vx = (float)(rand() % 21 - 10) / 10.0; // Random velocity between -1 and 1
      particles[i].vy = (float)(rand() % 21 - 10) / 10.0; // Random velocity between -1 and 1
    }
    for (int i = 0; i < 3; i++) {
        printf("Particle %d:\n", i + 1);
        printParticle(&particles[i]);
        for (int step = 1; step <= 5; step++) {
            updateParticle(&particles[i]);
            printf("After Step %d:\n", step);
            printParticle(&particles[i]);
        }
    }
    return 0;
}
```

- **Explanation of the program:**

The program simulates the behavior of particles in a two-dimensional space. It creates and tracks the movement of three particles over a series of five time steps. Let's break down the program and have explanation of its components:

Header Inclusions: These lines include standard C libraries necessary for input/output, dynamic memory allocation, and time functions.

Particle Structure:This defines a structure named `Particle` to represent a particle's properties. Each particle has two components: position (x and y) and velocity (vx and vy).

updateParticle Function: This function takes a pointer to a `Particle` structure as an argument and updates its position based on its current velocity. It effectively moves the particle by adding its velocity to its position.

printParticle Function: This function takes a pointer to a `Particle` structure and prints its position and velocity with two decimal places.

Main Function: `srand(time(NULL))` seeds the random number generator with the current time, ensuring that each run of the program produces different random values.

An array of four `Particle` structures is created to store particle data. However, in the subsequent code, only three particles are generated and simulated.

Inside a loop, random values for the position (x and y) and velocity (vx and vy) of three particles are generated and stored in the `particles` array. The positions are random values between 0 and 10, and the velocities are random values between -1 and 1.

Another loop iterates through the three particles. For each particle, it prints its initial information and then simulates its movement for five time steps, updating and printing its information after each step.

Simulation Loop: Inside the inner loop, the `updateParticle` function is called to update the particle's position based on its velocity. Then, the new particle information is printed, including the step number.

Return and End: Finally, the `main` function returns 0, indicating successful program execution, and the program ends.

In summary, this program generates and tracks the movement of three particles in a two-dimensional space over five time steps, printing their positions and velocities at each step. It demonstrates the use of structures, functions, and random number generation in a simple particle simulation.

- **Expected Results:**

```
Particle 1:
Position: (4.00, 7.90)
Velocity: (-0.40, -0.30)

After Step 1:
Position: (3.60, 7.60)
Velocity: (-0.40, -0.30)

After Step 2:
Position: (3.20, 7.30)
Velocity: (-0.40, -0.30)

After Step 3:
Position: (2.80, 7.00)
Velocity: (-0.40, -0.30)

After Step 4:
Position: (2.40, 6.70)
Velocity: (-0.40, -0.30)

After Step 5:
Position: (2.00, 6.40)
Velocity: (-0.40, -0.30)

Particle 2:
Position: (8.20, 2.50)
Velocity: (-0.20, -1.00)

After Step 1:
Position: (8.00, 1.50)
Velocity: (-0.20, -1.00)

After Step 2:
Position: (7.80, 0.50)
Velocity: (-0.20, -1.00)

After Step 3:
Position: (7.60, -0.50)
Velocity: (-0.20, -1.00)

After Step 4:
Position: (7.40, -1.50)
Velocity: (-0.20, -1.00)

After Step 5:
Position: (7.20, -2.50)
Velocity: (-0.20, -1.00)

Particle 3:
Position: (0.50, 3.70)
Velocity: (-0.50, 1.00)

After Step 1:
Position: (0.00, 4.70)
Velocity: (-0.50, 1.00)
```

```
After Step 2:
Position: (-0.50, 5.70)
Velocity: (-0.50, 1.00)

After Step 3:
Position: (-1.00, 6.70)
Velocity: (-0.50, 1.00)

After Step 4:
Position: (-1.50, 7.70)
Velocity: (-0.50, 1.00)

After Step 5:
Position: (-2.00, 8.70)
Velocity: (-0.50, 1.00)
```

8. Session – 3: Assignment - Tic-Tac-Toe Game:

- **Problem Statement:** Dynamic Memory Allocation, Arrays, and Structures: Tic-Tac-Toe Game

In this programming assignment, you will implement a Tic-Tac-Toe game in C that utilizes dynamic memory allocation, arrays, and structures. The game will be played between two players, 'X' and 'O', on a 3x3 grid. The program will dynamically allocate memory for the grid and use structures to represent the players and their moves.

1. Create a structure named Player with the following attributes:
    a. char symbol: To store the player's symbol ('X' or 'O').
    b. int score: To store the player's score (1 for win, 0 for loss).
2. Implement a function named initializeGrid that dynamically allocates memory for a 3x3 grid. Initialize the grid with empty spaces (' ').
3. Implement a function named printGrid that takes the dynamically allocated grid as input and prints the current state of the Tic-Tac-Toe grid.
4. Implement a function named checkWin that checks if a player has won the game. It should take the grid and the current player's symbol as input and return 1 if the player has won, 0 otherwise.
5. Implement the main function:
    a. Initialize two instances of the Player structure, one for 'X' and one for 'O'.
    b. Use a loop to alternate between players and allow them to make moves.
    c. For each move, prompt the current player to enter their row and column choices.
    d. Validate the inputs to ensure they are within the valid range and the chosen cell is not already occupied.
    e. After each move, update the grid and print its current state.
    f. Check if the current player has won using the checkWin function.
    g. If a player wins, update their score and display the winner.
    h. If the grid is full and no one has won, display a tie message.
6. Properly deallocate the dynamically allocated memory for the grid at the end of the program.
7. Use proper error handling and input validation to handle incorrect user inputs gracefully.

Note: This assignment focuses on dynamic memory allocation, arrays and structures. Students have to write the program in the observation book by their own effort and execute in the lab.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- Use Dynamic Memory Allocation in C to allocate arrays dynamically.
- Apply Control Structures and Functions in C programs.
- Create a structure in C and use them in application programs.
- Make use of pointers with structures in creating applications.

9. Session – 4: Lab Experiment - Doubly Linked List:

- **Problem Statement:**

A linked list is a linear data structure used in computer science and programming for storing and organizing a collection of elements. A linked list can be visualized as a chain of nodes, where each node points to the next node, creating a sequence. Create a C program on a doubly linked list that has a header node which maintains additional information of the list like – Number of nodes. Perform insertion at beginning, at end, at a position in the list, perform deletion at beginning, at end and of an item from the list, also include a display operation to display the contents of the list. Your program must be a menu driven program that provides choices to perform various operations. The node should contain the SRN, Name, branch and semester information of students in a college.

Note: In this exercise, you will practice implementing a doubly linked list.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- implement doubly linked list and use in various applications.
- distinguish between array and linked list.

- **Theory:**

A doubly linked list is a data structure in which each element (node) contains two references: one to the next node and another to the previous node in the sequence. This bidirectional linkage allows for efficient traversal in both forward and backward directions, making it useful for various applications like implementing dynamic data structures and algorithms.

- **Algorithm: Doubly Linked List (DLL)**

1. Structures:
   - Define structures for a student node (with data fields) and a header node to maintain the list's properties.
2. Create a Student Node:
   - Create a function createStudentNode() to allocate memory for a new student node, initialize its data fields, and set the previous and next pointers to NULL.
3. Get Student Data:
   - Create a function getStudentData() to input student data and return a new student node.
4. Insert at Front:
   - Create a function insertAtFront(header) to insert a student node at the front of the DLL.
     - Get student data using getStudentData().
     - If the list is empty, set the new student node as the first node.
     - Otherwise, update pointers to insert the new node at the front.
5. Display DLL:
   - Create a function display(header) to traverse and display the entire DLL from the header node.
6. Insert at End:
   - Create a function insertAtEnd(header) to insert a student node at the end of the DLL.
     - Get student data using getStudentData().
     - If the list is empty, set the new student node as the first node.
     - Otherwise, traverse to the end and update pointers to insert the new node.
7. Insert at Position:
   - Create a function insertAtPosition(header, position) to insert a student node at a specified position.
     - Validate the position.
     - If the position is 1, use insertAtFront().
     - If the position is at the end, use insertAtEnd().
     - Otherwise, find the position and update pointers accordingly.
8. Delete from Front:
   - Create a function deleteFromFront(header) to delete a student node from the front.
     - Handle the case when the list is empty.
     - Update pointers and free memory.
9. Delete from End:
   - Create a function deleteFromEnd(header) to delete a student node from the end.
     - Handle cases for an empty list and a single-node list.

- Traverse to the end, update pointers, and free memory.

10. Delete by SRN:
    - Create a function deleteBySRN(header, srn) to delete a student node based on the SRN (Student Registration Number).
      - Search for the SRN in the list.
      - Handle cases for not finding the SRN, and deleting the first and other nodes.
11. Main Function:
    - Create a main() function to interact with the user, present a menu of options, and call the appropriate functions based on the user's choice.
12. Free Memory:
    - Before exiting the program, free the memory allocated for all student nodes.
13. Exit:
    - Implement an exit option in the menu to gracefully exit the program.
14. Error Handling:
    - Include error handling for invalid inputs, such as invalid positions or non-existent SRNs.
15. Loop:
    - Use a loop (e.g., a do-while loop) in the main() function to repeatedly display the menu until the user chooses to exit.

- **Source Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Structure to hold student data
struct Student {
   char srn[20];
   char name[50];
   char branch[50];
   int sem;
   struct Student* prev;
   struct Student* next;
};
// Structure for header node
struct Header {
   int numStudents;
   struct Student* first;
};
// Function to create a new student node
struct Student* createStudentNode() {
   struct Student* newStudent = (struct Student*)malloc(sizeof(struct Student));
   newStudent->prev = NULL;
   newStudent->next = NULL;
   return newStudent;
}
// Function to get Student data
struct Student* getStudentData() {
   struct Student* newStudent = createStudentNode();
   printf("Enter SRN: ");
   scanf("%s", newStudent->srn);
   printf("Enter Name: ");
   scanf(" %[^\n]s", newStudent->name);
   printf("Enter Branch: ");
   scanf(" %[^\n]s", newStudent->branch);
   printf("Enter Semester: ");
   scanf("%d", &newStudent->sem);
   return newStudent;
}
// Function to insert a student at the front of the DLL
void insertAtFront(struct Header* header) {
   struct Student* newStudent = getStudentData();
```

```
    if (header->first == NULL) {
        header->first = newStudent;
    } else {
        newStudent->next = header->first;
        header->first->prev = newStudent;
        header->first = newStudent;
    }
    header->numStudents++;
    printf("Student inserted at the front.\n");
}
// Function to display the DLL
void display(struct Header* header) {
    if (header->first == NULL) {
        printf("DLL is empty.\n");
        return;
    }
    printf("Number of students: %d\n", header->numStudents);
    printf("SRN\tBranch\tSemester\tName\n");
    struct Student* current = header->first;
    while (current != NULL) {
        printf("%s\t%s\t%d\t\t%s\n", current->srn, current->branch, current->sem, current->name);
        current = current->next;
    }
}
// Function to insert a student at the end of the DLL
void insertAtEnd(struct Header* header) {
    struct Student* newStudent = getStudentData();
    if (header->first == NULL) {
        header->first = newStudent;
    } else {
        struct Student* current = header->first;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newStudent;
        newStudent->prev = current;
    }
    header->numStudents++;
    printf("Student inserted at the end.\n");
}
// Function to insert a student at a given position in the DLL
void insertAtPosition(struct Header* header, int position) {
    if (position < 1 || position > header->numStudents + 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        insertAtFront(header);
        return;
    }
    if (position == header->numStudents + 1) {
        insertAtEnd(header);
        return;
    }
    struct Student* newStudent = getStudentData();
    struct Student* current = header->first;
    for (int i = 1; i < position - 1; i++) {
        current = current->next;
    }
    newStudent->prev = current;
```

```
    newStudent->next = current->next;
    current->next->prev = newStudent;
    current->next = newStudent;
    header->numStudents++;
    printf("Student inserted at position %d.\n", position);
}
// Function to delete a student from the front of the DLL
void deleteFromFront(struct Header* header) {
    if (header->first == NULL) {
        printf("DLL is empty.\n");
        return;
    }
    struct Student* temp = header->first;
    header->first = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = NULL;
    }
    free(temp);
    header->numStudents--;
    printf("Student deleted from the front.\n");
}
// Function to delete a student from the end of the DLL
void deleteFromEnd(struct Header* header) {
    if (header->first == NULL) {
        printf("DLL is empty.\n");
        return;
    }
    if (header->first->next == NULL) {
        free(header->first);
        header->first = NULL;
    } else {
        struct Student* current = header->first;
        while (current->next != NULL) {
            current = current->next;
        }
        current->prev->next = NULL;
        free(current);
    }
    header->numStudents--;
    printf("Student deleted from the end.\n");
}
// Function to delete a student based on SRN
void deleteBySRN(struct Header* header, const char* srn) {
    if (header->first == NULL) {
        printf("DLL is empty.\n");
        return;
    }
    struct Student* current = header->first;
    while (current != NULL) {
        if (strcmp(current->srn, srn) == 0) {
            if (current->prev == NULL) {
                header->first = current->next;
            } else {
                current->prev->next = current->next;
            }
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            free(current);
            header->numStudents--;
```

```
            printf("Student with SRN %s deleted.\n", srn);
            return;
        }
        current = current->next;
    }
    printf("Student with SRN %s not found.\n", srn);
}
int main() {
    struct Header header;
    header.numStudents = 0;
    header.first = NULL;
    int choice;
    char srnToDelete[20];
    do {
        printf("\nMenu:\n");
        printf("1. Insert at Front\n");
        printf("2. Display DLL\n");
        printf("3. Insert at End\n");
        printf("4. Insert at Position\n");
        printf("5. Delete from Front\n");
        printf("6. Delete from End\n");
        printf("7. Delete by SRN\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insertAtFront(&header);
                break;
            case 2:
                display(&header);
                break;
            case 3:
                insertAtEnd(&header);
                break;
            case 4:
                {
                    int position;
                    printf("Enter position: ");
                    scanf("%d", &position);
                    insertAtPosition(&header, position);
                }
                break;
            case 5:
                deleteFromFront(&header);
                break;
            case 6:
                deleteFromEnd(&header);
                break;
            case 7:
                printf("Enter SRN to delete: ");
                scanf("%s", srnToDelete);
                deleteBySRN(&header, srnToDelete);
                break;
            case 8:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
```

```
        }
    } while (choice != 8);
    // Free allocated memory before exiting
    struct Student* current = header.first;
    while (current != NULL) {
        struct Student* temp = current;
        current = current->next;
        free(temp);
    }
    return 0;
}
```

- **Explanation of the program:**

This program demonstrates the implementation of a Doubly Linked List (DLL) to manage student records. Here's an explanation:

1. Data Structures:
    - struct Student: This structure represents a student and contains fields for SRN (Student Roll Number), name, branch, semester, and two pointers (prev and next) for the previous and next students in the DLL.
    - struct Header: This structure serves as the header node for the DLL and contains the total number of students (numStudents) and a pointer to the first student (first) in the list.
2. Functions:
    - createStudentNode(): This function dynamically allocates memory for a new student node and initializes its prev and next pointers, returning the created node.
    - getStudentData(): This function collects student data (SRN, name, branch, and semester) from the user and returns a new student node containing this data.
    - insertAtFront(struct Header* header): Inserts a new student at the front of the DLL. If the DLL is empty, it becomes the first student; otherwise, it pushes the existing students forward.
    - display(struct Header* header): Displays the contents of the DLL, including the number of students, their SRNs, branches, semesters, and names.
    - insertAtEnd(struct Header* header): Inserts a new student at the end of the DLL. If the DLL is empty, the new student becomes the first one; otherwise, it's added after the current last student.
    - insertAtPosition(struct Header* header, int position): Inserts a student at a specified position in the DLL. This function handles inserting at the front or end and adjusts the pointers accordingly.
    - deleteFromFront(struct Header* header): Removes a student from the front of the DLL and adjusts pointers accordingly.
    - deleteFromEnd(struct Header* header): Removes a student from the end of the DLL and adjusts pointers accordingly.
    - deleteBySRN(struct Header* header, const char* srn): Deletes a student based on their SRN (Student Roll Number) by searching for the SRN in the DLL.
3. Main Function (main()):
    - Initializes the DLL header node and variables.
    - Provides a menu-driven interface for the user to perform various operations on the DLL, such as inserting, deleting, displaying, or exiting the program.
    - The program continues to display the menu and execute the chosen operation until the user selects the exit option.
    - Properly handles memory deallocation by freeing all allocated student nodes before exiting the program.

- **Expected Results:**

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 1
Enter SRN: SRN001
Enter Name: John Doe
Enter Branch: Computer Science
Enter Semester: 3
Student inserted at the front.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 1
Enter SRN: SRN002
Enter Name: Jane Smith
Enter Branch: Electrical Engineering
Enter Semester: 2
Student inserted at the front.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 2
Number of students: 2
SRN Branch Semester Name
SRN002 Electrical Engineering 2 Jane Smith
SRN001 Computer Science 3 John Doe

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End

4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 3
Enter SRN: SRN003
Enter Name: Bob Johnson
Enter Branch: Mechanical Engineering
Enter Semester: 4
Student inserted at the end.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 2
Number of students: 3
SRN Branch Semester Name
SRN002 Electrical Engineering 2 Jane Smith
SRN001 Computer Science 3 John Doe
SRN003 Mechanical Engineering 4 Bob Johnson

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 4
Enter position: 2
Enter SRN: SRN004
Enter Name: Sarah Wilson
Enter Branch: Civil Engineering
Enter Semester: 5
Student inserted at position 2.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End

7. Delete by SRN
8. Exit
Enter your choice: 2
Number of students: 4
SRN Branch Semester Name
SRN002 Electrical Engineering 2 Jane Smith
SRN004 Civil Engineering 5 Sarah Wilson
SRN001 Computer Science 3 John Doe
SRN003 Mechanical Engineering 4 Bob Johnson

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 5
Student deleted from the front.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 6
Student deleted from the end.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 7
Enter SRN to delete: SRN001
Student with SRN SRN001 deleted.

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position

5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 2
Number of students: 2
SRN Branch Semester Name
SRN004 Civil Engineering 5 Sarah Wilson
SRN003 Mechanical Engineering 4 Bob Johnson

Menu:
1. Insert at Front
2. Display DLL
3. Insert at End
4. Insert at Position
5. Delete from Front
6. Delete from End
7. Delete by SRN
8. Exit
Enter your choice: 8
Exiting...

10. Session – 5: Lab Experiment - Implementing a Stack using Linked List:

- **Problem Statement:**

You are required to implement all the basic stack operations i.e., push, pop, peek, isEmpty, and display.
The following are the rubrics to be followed while preparing the code.

1. **Data Structures:**
   a. Define a structure Node to represent each element of the linked list. It should contain two fields: a data field to store the value, and a pointer to the next node.
   b. Define a structure Stack to represent the stack. It should contain a pointer to the top node of the linked list.
2. **Initialization:**
   a. Implement a function void initialize(Stack *stack) to initialize the stack. This function should set the top pointer to NULL.
3. **Push Operation:**
   a. Implement a function void push(Stack *stack, int value) to push an element onto the stack.
   b. Create a new node with the provided value and link it to the top of the stack.
4. **Pop Operation:**
   a. Implement a function int pop(Stack *stack) to pop an element from the stack.
   b. Return the value of the top node, update the top pointer, and free the memory of the popped node.
5. Peek Operation:
   a. Implement a function int peek(Stack *stack) to retrieve the value of the top element without removing it.
   b. Return the value of the top node without modifying the stack.
6. **isEmpty Operation:**
   a. Implement a function int isEmpty(Stack *stack) to check if the stack is empty.
   b. Return 1 if the stack is empty, and 0 otherwise.
7. **Display Operation:**
   a. Implement a function void display(Stack *stack) to display the elements of the stack from top to bottom.
   b. Traverse through the linked list and print each element's value.
8. **Testing:**
   a. Create a menu driven main function to test your stack implementation.
   b. Perform a sequence of push, pop, and peek operations to demonstrate the functionality of the stack.
   c. Test edge cases, such as popping from an empty stack and peeking into an empty stack, to ensure proper error handling.

Implement error handling for edge cases, such as stack overflow and underflow. For example, if the stack is full and a push operation is attempted, or if a pop operation is attempted on an empty stack, display appropriate error messages.
Note: In this exercise, you will practice implementing a stack data structure using a linked list.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- infer a practical understanding of how fundamental data structures like stacks can be implemented using more complex data structures like linked lists.
- develop problem-solving skills and learn to handle edge cases and error conditions in a real-world programming context, improving their overall programming proficiency.

- **Theory:**

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. It is a collection of elements with two primary operations:

1. Push: This operation adds an element to the top of the stack.
2. Pop: This operation removes and returns the top element from the stack.

Additional stack operations may include:

3. Peek (or Top): Retrieves the top element without removing it from the stack.
4. isEmpty: Checks if the stack is empty.

Stacks are commonly used for tasks like managing function calls in a program (call stack), parsing expressions, and tracking state in algorithms. They provide a simple and efficient way to manage and access data in a last-in, first-out manner.

- **Algorithm - Stack with Linked List:**

1. Define a structure Node to represent each element of the linked list. It should contain two fields: data and next (pointer to the next node).
2. Define a structure Stack to represent the stack. It should contain a pointer to the top node.
3. Initialize the stack by setting the top pointer to NULL.

Stack Initialization:
- Procedure: Initialize(Stack *stack)
- Input: Pointer to the Stack structure
- Output: None
- Implementation:
  - Set stack's top pointer to NULL.

Push Operation:
- Procedure: Push(Stack *stack, int value)
- Input: Pointer to the Stack structure, value to be pushed
- Output: None
- Implementation:
  - Create a new Node with the provided value.
  - Set the new Node's next pointer to point to the current top node (if any).
  - Update the top pointer of the stack to point to the new Node.

Pop Operation:
- Procedure: int Pop(Stack *stack)
- Input: Pointer to the Stack structure
- Output: The value of the popped element (or an error value if the stack is empty)
- Implementation:
  - If the stack is empty (top pointer is NULL), return an error value (e.g., -1).
  - Otherwise, store the value of the top Node.
  - Update the top pointer of the stack to point to the next Node.
  - Free the memory of the popped Node.
  - Return the stored value.

Peek Operation:
- Procedure: int Peek(Stack *stack)
- Input: Pointer to the Stack structure
- Output: The value of the top element (or an error value if the stack is empty)
- Implementation:
  - If the stack is empty (top pointer is NULL), return an error value (e.g., -1).
  - Otherwise, return the value of the top Node.

isEmpty Operation:
- Procedure: int isEmpty(Stack *stack)
- Input: Pointer to the Stack structure
- Output: 1 if the stack is empty, 0 otherwise
- Implementation:
  - If the stack's top pointer is NULL, return 1 (indicating empty); otherwise, return 0.

Display Operation:
- Procedure: Display(Stack *stack)
- Input: Pointer to the Stack structure
- Output: None
- Implementation:
  - Start from the top of the stack.
  - Traverse through the linked list and print each element's value.

- **Source Code:**

```
#include <stdio.h>
#include <stdlib.h>
// Define the Node structure
struct Node {
   int data;
   struct Node* next;
};
```

```
// Define the Stack structure
struct Stack {
    struct Node* top;
};
// Initialize the stack
void initialize(struct Stack* stack) {
    stack->top = NULL;
}
// Push operation
void push(struct Stack* stack, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = value;
    newNode->next = stack->top;
    stack->top = newNode;
    printf("%d pushed onto the stack.\n", value);
}
// Pop operation
int pop(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack underflow.\n");
        return -1;
    }
    struct Node* temp = stack->top;
    int value = temp->data;
    stack->top = temp->next;
    free(temp);
    return value;
}
// Peek operation
int peek(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty.\n");
        return -1;
    }
    return stack->top->data;
}
// Check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == NULL;
}
// Display the stack
void display(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* current = stack->top;
    printf("Stack elements from top to bottom:\n");
    while (current != NULL) {
        printf("%d\n", current->data);
        current = current->next;
    }
}
int main() {
    struct Stack stack;
    initialize(&stack);
```

```
    int choice, value;
    do {
        printf("\nStack Menu:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stack, value);
                break;
            case 2:
                value = pop(&stack);
                if (value != -1)
                    printf("Popped value: %d\n", value);
                break;
            case 3:
                value = peek(&stack);
                if (value != -1)
                    printf("Top value: %d\n", value);
                break;
            case 4:
                display(&stack);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}
```

- **Explanation of the program:**

This program defines and implements a basic stack data structure using a singly linked list. Here's a brief explanation of the code:

1. Structures:
    - Two structures are defined: struct Node to represent individual elements of the stack (nodes) and struct Stack to represent the stack itself. Each node contains an integer value (data) and a pointer to the next node (next).
2. Initialization:
    - The initialize function initializes an empty stack by setting its top pointer to NULL.
3. Push Operation:
    - The push function adds a new node with the specified integer value to the top of the stack. It dynamically allocates memory for the new node, assigns the value, and updates the top pointer to point to the new node. It also handles memory allocation failure.
4. Pop Operation:
    - The pop function removes and returns the top element from the stack. It checks if the stack is empty (underflow) and, if not, it removes the top node, frees the memory, and returns the value. It also handles stack underflow.
5. Peek Operation:

- The peek function returns the value of the top element of the stack without removing it. It checks if the stack is empty and, if not, returns the value.

6. isEmpty Function:
   - The isEmpty function checks if the stack is empty by examining whether the top pointer is NULL. It returns 1 if the stack is empty and 0 otherwise.
7. Display Function:
   - The display function prints the elements of the stack from top to bottom. It checks if the stack is empty and, if not, iterates through the linked list, printing each element.
8. Main Function:
   - The main function serves as a user interface to interact with the stack:
     - It initializes an empty stack.
     - It presents a menu with options for pushing, popping, peeking, displaying, and exiting.
     - Depending on the user's choice, it calls the appropriate stack operation functions.
     - The menu continues to be displayed until the user chooses to exit.

- **Expected Results:**

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 10
10 pushed onto the stack.

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 20
20 pushed onto the stack.

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements from top to bottom:
20
10

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2

```
Popped value: 20

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top value: 10

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped value: 10

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack is empty.

Stack Menu:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting...
```

11. Session – 6: Assignment - Circular Queue Implementation with Linked Storage:

- **Problem Statement:**

You have been tasked with implementing a circular queue data structure using linked storage in C language. The queue should include a header node that stores the number of nodes in the queue, as well as the largest and smallest values currently present in the queue. Implement the following operations for this circular queue:

1. Initialize Queue: Write a function to initialize an empty circular queue with a header node. The header node should have its count initialized to 0, and largest and smallest initialized to appropriate values.
2. Enqueue: Write a function to enqueue an integer value into the circular queue. Update the header node's count, largest, and smallest accordingly.
3. Dequeue: Write a function to dequeue an element from the circular queue. Update the header node's count, largest, and smallest appropriately.
4. Get Largest Value: Write a function to retrieve the largest value from the circular queue.
5. Get Smallest Value: Write a function to retrieve the smallest value from the circular queue.
6. Display Queue: Write a function to display the elements of the circular queue, starting from the front.

Note: In this exercise, you will practice implementing a queue data structure using a linked list.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- Demonstrate the ability to apply Circular Queue implementation with linked storage to efficiently manage data by successfully enqueueing and dequeueing elements in a cyclic manner.
- Evaluate the Circular Queue implementation with linked storage to identify potential areas for improvement, such as optimizing storage utilization or enhancing error-handling mechanisms.

12. Session – 7: Lab Experiment - Infix to Postfix Conversion using Dynamic Stack:

- **Problem Statement:**

Write a C program to convert an infix expression containing the operators +, -, *, /, %, and ^, along with parentheses, into its postfix form. Your program should use a dynamic flexible array to implement the stack for the conversion process. The program should also be case-insensitive, meaning that it should recognize operators and operands regardless of their case (upper or lower).
Your program should perform the following steps:

1. Read an infix expression from the user.
2. Convert the infix expression to postfix using the dynamic stack.
3. Display the resulting postfix expression.

Remember to handle operator precedence and associativity correctly during the conversion process.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- Analyze and apply the principles of dynamic stack data structures to successfully convert infix expressions to postfix notation, demonstrating an understanding of the algorithm's step-by-step execution.
- Evaluate the efficiency and correctness of infix to postfix conversion using dynamic stack operations through problem-solving and debugging skills, ensuring accurate results for complex mathematical expressions.

- **Theory:**

Infix notation is the typical way we write arithmetic expressions, where operators are placed between operands, such as "3 + 5 * 2." Postfix notation, also known as Reverse Polish Notation (RPN), represents the same expression as "3 5 2 * +". Converting infix expressions to postfix notation is often required for efficient evaluation and parsing.
Dynamic stacks can be particularly useful for implementing this conversion algorithm as they allow for dynamic resizing during the process.
Here's a short theory for performing infix to postfix conversion using a dynamic stack:

1. Initialize an empty dynamic stack to hold operators and a string to store the postfix expression.
2. Scan the infix expression from left to right, character by character.
3. For each character in the input expression:
    - If it is an operand (a digit or a letter), append it to the postfix string.
    - If it is an operator (+, -, *, /, etc.), perform the following steps: a. While the stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator at the top of the stack, pop the top operator from the stack and append it to the postfix string. b. Push the current operator onto the stack.
4. If you encounter an opening parenthesis '(', push it onto the stack.
5. If you encounter a closing parenthesis ')', pop operators from the stack and append them to the postfix string until you encounter the corresponding opening parenthesis '('. Pop and discard the '('.
6. Continue this process until you have scanned the entire input infix expression.
7. After processing all the characters, if there are any operators left in the stack, pop them and append them to the postfix string.
8. The postfix string now contains the converted expression in postfix notation.

By using a dynamic stack, you can handle expressions of varying lengths without worrying about stack overflow or inefficient memory usage. The dynamic stack automatically resizes itself as needed, making this algorithm versatile and suitable for practical implementations of infix to postfix conversion.

- **Algorithm:**

Converting an infix expression to postfix notation can be accomplished using a dynamic stack (a stack that grows or shrinks as needed). The algorithm typically involves scanning the infix expression from left to right and using a stack to keep track of operators and operands. Here's an algorithm for infix to postfix conversion using a dynamic stack:

1. Initialize an empty dynamic stack to store operators.
2. Initialize an empty list or string to store the postfix expression.
3. Scan the infix expression from left to right, one character at a time.
    - If the character is an operand (a digit or variable), append it to the postfix expression list.
    - If the character is an open parenthesis '(', push it onto the stack.
    - If the character is a close parenthesis ')', pop operators from the stack and append them to the postfix expression list until an open parenthesis '(' is encountered. Pop and discard the open parenthesis.

- If the character is an operator (+, -, *, /, etc.), do the following: a. While the stack is not empty and the precedence of the operator at the top of the stack is greater than or equal to the precedence of the current operator, pop the operator from the stack and append it to the postfix expression list. b. Push the current operator onto the stack.

4. After scanning the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression list.
5. The postfix expression list now contains the converted expression.

- **Source Code:**

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

#define MAX_SIZE 100

typedef struct {
    char *arr;
    int top;
    int capacity;
} Stack;

void initialize(Stack *stack, int capacity) {
    stack->arr = (char *)malloc(sizeof(char) * capacity);
    stack->top = -1;
    stack->capacity = capacity;
}
int isEmpty(Stack *stack) {
    return stack->top == -1;
}
int isFull(Stack *stack) {
    return stack->top == stack->capacity - 1;
}
void push(Stack *stack, char item) {
    if (isFull(stack)) {
        printf("Stack overflow!\n");
        exit(1);
    }
    stack->arr[++stack->top] = item;
}
char pop(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        exit(1);
    }
    return stack->arr[stack->top--];
}
char peek(Stack *stack) {
    if (isEmpty(stack)) {
        return '\0';
    }
    return stack->arr[stack->top];
}
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/' || op == '%') return 2;
    if (op == '^') return 3;
    return 0;
}
void infixToPostfix(char *infix, char *postfix) {
```

```
    Stack stack;
    initialize(&stack, MAX_SIZE);
    int i, j;
    char c, temp;

    for (i = 0, j = 0; infix[i] != '\0'; i++) {
        c = infix[i];
        if (isalpha(c) || isdigit(c)) {
            postfix[j++] = c;
        } else if (c == '(') {
            push(&stack, c);
        } else if (c == ')') {
            while ((temp = pop(&stack)) != '(') {
                postfix[j++] = temp;
            }
        } else {
            while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(c)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, c);
        }
    }
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }

    postfix[j] = '\0';
}
int main() {
    char infix[MAX_SIZE], postfix[MAX_SIZE];
    printf("Enter an infix expression: ");
    fgets(infix, sizeof(infix), stdin);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

- **Explanation of the program:**

This program is designed to convert an infix mathematical expression entered by the user into its equivalent postfix notation using a stack data structure. Here's a brief explanation of the program:

1. **Stack Implementation:** The program defines a custom stack data structure using a structure called `Stack`. It includes functions to initialize, check if the stack is empty or full, push elements onto the stack, pop elements from the stack, and peek at the top element.
2. **Operator Precedence:** The program defines a function `precedence()` to assign precedence values to operators. This is used later to determine the order of operators in the postfix expression.
3. **Infix to Postfix Conversion:** The `infixToPostfix()` function takes an infix expression as input and converts it to postfix notation. It uses a stack to handle operators and operands while iterating through the infix expression character by character. The conversion process follows these rules:
    - Operand characters (alphabets and digits) are appended directly to the postfix expression.
    - An open parenthesis '(' is pushed onto the stack.
    - When a close parenthesis ')' is encountered, operators are popped from the stack and added to the postfix expression until an open parenthesis '(' is encountered. The open parenthesis is also popped and discarded.
    - For operators (+, -, *, /, %, ^), the program checks their precedence relative to the operators already on the stack. Operators with higher precedence should be pushed onto the stack first. The stack is emptied of higher or equal precedence operators before the current operator is pushed.
4. **Main Function:** In the `main()` function, the program takes an infix expression as input from the user using `fgets()`, then calls the `infixToPostfix()` function to convert it to postfix notation. Finally, it prints the resulting postfix expression.

- **Expected Results:**

Enter an infix expression: (A+B)*(C-D)
Postfix expression: AB+CD-*

13. Session – 8: Lab Experiment - Evaluating Postfix Expressions using Linked List-based Stack:

- **Problem Statement:**

Write a C program to evaluate a postfix expression using a linked list to implement the stack. The program should validate inputs to ensure that only numbers and valid operators are accepted.
Your program should perform the following steps:

1. Define a structure for a linked list node that holds an integer value and a pointer to the next node.
2. Implement functions to perform stack operations (push, pop, isEmpty, etc.) using the linked list.
3. Implement a function to evaluate a postfix expression. The function should accept a string containing the postfix expression and return the result of the evaluation.
4. The program should validate the input expression to ensure that only numbers and valid operators (+, -, *, /, %) are present. Any other characters should be considered invalid input.
5. Display appropriate error messages if the input expression is invalid or if there's a problem during evaluation.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- critique the efficiency of a Linked List-based Stack in evaluating postfix expressions, identifying its strengths and weaknesses.
- appraise the accuracy of their postfix expression evaluation using a Linked List-based Stack through critical analysis and error identification.

- **Theory:**

Postfix notation, also known as Reverse Polish Notation (RPN), is a mathematical notation in which operators follow their operands. Evaluating postfix expressions involves processing mathematical expressions without the need for parentheses or a specific order of operations. Here's a short theory for evaluating postfix expressions:

1. **Operand-Operator Relationship**: In postfix notation, operands are placed before the operators that act upon them. This characteristic simplifies the evaluation process as it eliminates the need to consider operator precedence and parentheses.
2. **Stack-Based Evaluation**: The key to evaluating postfix expressions is the use of a stack data structure. The stack allows us to store operands until their corresponding operator is encountered. The evaluation process can be summarized as follows:
    - Initialize an empty stack.
    - Start scanning the postfix expression from left to right.
    - For each element (either an operand or operator):
        - If it's an operand, push it onto the stack.
        - If it's an operator, pop the necessary number of operands (two for binary operators) from the stack, perform the operation, and push the result back onto the stack.
    - Continue this process until you've scanned the entire expression.
    - The final result will be on top of the stack.
3. **Advantages of Postfix Notation**:
    - Eliminates the need for parentheses and explicit operator precedence.
    - Simplifies expression evaluation by following a strict left-to-right parsing.
    - Well-suited for use in calculators and compilers for its simplicity.

- **Algorithm - Evaluate Postfix Expression:**

1. Initialize an empty stack to hold operands.
2. Split the postfix expression into tokens (operands and operators). You can use whitespace as a delimiter to separate tokens.
3. Iterate through each token in the postfix expression: a. If the token is an operand, push it onto the stack. b. If the token is an operator: i. Pop the top two operands from the stack. ii. Perform the operation on these operands. iii. Push the result back onto the stack.
4. After processing all tokens, the stack should contain only one element, which is the final result of the expression. Pop this result from the stack.

- **Source Code:**

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

struct Node {
   int data;
   struct Node* next;
};
typedef struct Node Node;

struct Stack {
   Node* header;
};
typedef struct Stack Stack;

// Function to initialize the stack
void initializeStack(Stack* stack) {
   stack->header = NULL;
}
// Function to push a value onto the stack
void push(Stack* stack, int value) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      exit(EXIT_FAILURE);
   }
   newNode->data = value;
   newNode->next = stack->header;
   stack->header = newNode;
}
// Function to pop a value from the stack
int pop(Stack* stack) {
   if (stack->header == NULL) {
      printf("Stack underflow\n");
      exit(EXIT_FAILURE);
   }
   Node* temp = stack->header;
   int value = temp->data;
   stack->header = temp->next;
   free(temp);
   return value;
}
// Function to check if the stack is empty
int isEmpty(Stack* stack) {
   return stack->header == NULL;
}
// Function to evaluate a postfix expression
int evaluatePostfix(char* expression) {
   Stack stack;
   initializeStack(&stack);

   for (int i = 0; expression[i] != '\0'; i++) {
      if (isdigit(expression[i])) {
         int num = 0;
         while (isdigit(expression[i])) {
            num = num * 10 + (expression[i] - '0');
            i++;
```

```
            }
            i--;  // Adjust for the extra increment in the loop
            push(&stack, num);
        } else if (expression[i] == ' ' || expression[i] == '\n') {
            continue;
        } else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '*' ||
                   expression[i] == '/' || expression[i] == '%') {
            if (isEmpty(&stack)) {
                printf("Invalid expression\n");
                exit(EXIT_FAILURE);
            }
            int operand2 = pop(&stack);
            if (isEmpty(&stack)) {
                printf("Invalid expression\n");
                exit(EXIT_FAILURE);
            }
            int operand1 = pop(&stack);
            int result;
            switch (expression[i]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    if (operand2 == 0) {
                        printf("Division by zero\n");
                        exit(EXIT_FAILURE);
                    }
                    result = operand1 / operand2;
                    break;
                case '%':
                    if (operand2 == 0) {
                        printf("Modulo by zero\n");
                        exit(EXIT_FAILURE);
                    }
                    result = operand1 % operand2;
                    break;
            }
            push(&stack, result);
        } else {
            printf("Invalid character in expression\n");
            exit(EXIT_FAILURE);
        }
    }
    if (isEmpty(&stack)) {
        printf("Invalid expression\n");
        exit(EXIT_FAILURE);
    }
    int finalResult = pop(&stack);
    if (!isEmpty(&stack)) {
        printf("Invalid expression\n");
        exit(EXIT_FAILURE);
    }
    return finalResult;
}
```

```
int main() {
   char expression[100];
   printf("Enter a postfix expression: ");
   fgets(expression, sizeof(expression), stdin);
   // Evaluate the expression
   int result = evaluatePostfix(expression);
   printf("Result: %d\n", result);
   return 0;
}
```

- **Explanation of the program:**

This code is an implementation of a program that evaluates postfix expressions using a stack data structure. Postfix notation represents mathematical expressions in a way that eliminates the need for parentheses and follows a strict order of operations. The code takes a postfix expression as input from the user and calculates its result.
Let's break down the code and provide an explanation for each part:

1. **Data Structures**:
   - `struct Node`: Defines a structure for a singly-linked list node, which will be used to implement the stack.
   - `struct Stack`: Defines a structure for the stack, which is essentially a linked list of nodes.
2. **Initialization Functions**:
   - `initializeStack(Stack* stack)`: Initializes an empty stack by setting the header pointer to `NULL`.
3. **Stack Manipulation Functions**:
   - `push(Stack* stack, int value)`: Pushes a value onto the stack by creating a new node and adding it to the top of the stack.
   - `pop(Stack* stack)`: Pops (removes and returns) a value from the stack. It also handles cases where the stack is empty (underflow).
   - `isEmpty(Stack* stack)`: Checks if the stack is empty by examining the header pointer.
4. **Expression Evaluation Function**:
   - `evaluatePostfix(char* expression)`: This is the core function for evaluating postfix expressions.
     - It initializes an empty stack.
     - It iterates through each character in the input expression.
     - If it encounters a digit (operand), it collects all contiguous digits to form the complete number and pushes it onto the stack.
     - If it encounters an operator (+, -, *, /, %), it pops the top two values from the stack, performs the corresponding operation, and pushes the result back onto the stack.
     - It handles division by zero and modulo by zero errors.
     - If it encounters an invalid character, it prints an error message and exits.
     - After processing all characters, it should have only one value left on the stack, which is the final result of the expression.
     - It checks for invalid expressions (e.g., too few operands, too few operators) and prints an error if necessary.
5. **Main Function**:
   - `main()`: The main function is responsible for getting the postfix expression as input from the user, calling the `evaluatePostfix` function to calculate the result, and printing the result.

- **Expected Results:**

Enter a postfix expression: 3 4 + 2 * 7 /
Result: 3

14. Session – 9: Assignment - Round Robin Scheduling Simulation – Queues:

- **Problem Statement:**

You have been tasked with implementing a round robin scheduling simulation using a dynamically allocated array-based queue in C. Your goal is to help manage a set of processes and simulate their execution based on the round robin scheduling algorithm, assuming time of arrival is same for all processes.
Write a C program that accomplishes the following tasks:

1. Prompt the user to enter the number of processes.
2. For each process, prompt the user to enter the burst time.
3. Prompt the user to enter the time quantum for the round robin scheduling.
4. Implement the round robin scheduling algorithm using a dynamically allocated array-based queue.
5. Display a Gantt chart that shows the execution order of processes along with their respective start and end times.
6. Ensure that the queue is properly managed for task scheduling.
7. Free any dynamically allocated memory before the program exits.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- analyze the impact of different time quantum values on the efficiency and fairness of task execution within the scheduling algorithm.
- create a detailed report that outlines the advantages and disadvantages of Round Robin Scheduling in comparison to other scheduling algorithms, substantiating their conclusions with empirical data from the simulation.

15. Session – 10: Lab Experiment - Complete Binary Tree:

- **Problem Statement:**

You are tasked with implementing a program that creates a complete binary tree from user input and offers different traversal options. Complete binary trees are special binary trees where all levels are completely filled except possibly the last level, and nodes are added from left to right.
Your program should perform the following steps:

1. Take input from the user for the number of elements `n` and then `n` elements to create a complete binary tree.
2. Implement a function to insert elements into the complete binary tree in level order.
3. Implement functions to perform inorder, preorder, and postorder traversals of the created binary tree.
4. Provide a menu-driven main function that allows the user to choose from the following options:
   a. Inorder Traversal
   b. Preorder Traversal
   c. Postorder Traversal
   d. Exit

Your implementation should create the complete binary tree and allow the user to select a traversal method from the menu. Each traversal method should display the corresponding order of node values.
Write a complete C program to achieve the above functionality. You are free to use any standard C libraries and programming constructs.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

1. apply their knowledge of Complete Binary Trees to efficiently perform insertion and deletion operations, ensuring that the tree remains complete throughout the process.
2. analyze various algorithms used to validate whether a given binary tree is a complete binary tree or not, demonstrating a deep understanding of the underlying properties and requirements for completeness in binary trees.

- **Theory:**

A complete binary tree is a special type of binary tree that exhibits a specific structural property. In a complete binary tree:

1. Node Distribution: All levels of the tree are completely filled from left to right, except possibly for the last level, which is filled from left to right as well but may have some missing nodes towards the right.
2. Node Count: The number of nodes at each level increases exponentially. Specifically, the number of nodes at level $'d'$ is 2^{d-1}, where $'d'$ is the depth or level of the tree.
3. Balanced Structure: The tree is inherently balanced, and its height (depth) is minimized for a given number of nodes. The height of a complete binary tree with 'n' nodes is at most $\lfloor \log_2 n + 1 \rfloor$, where $\lfloor x \rfloor$ represents the floor function.

Properties and Implications:

1. Efficient Storage: Complete binary trees are efficient for storage when implemented using an array-based representation. Due to their predictable structure, nodes can be stored compactly in an array without wasting space.
2. Easy Navigation: Navigation and indexing are straightforward in a complete binary tree when using an array-based representation. Parent-child relationships can be determined using simple arithmetic operations, making it efficient for various algorithms and data structures.
3. Insertion and Deletion: Insertion and deletion operations in a complete binary tree are typically efficient and can be managed to maintain the complete property. For example, when inserting a new node, it is usually added as the next available node in the last level from left to right.
4. Heaps: Complete binary trees are commonly used as the underlying structure for binary heaps, such as max heaps and min heaps, due to their balanced nature. This makes them suitable for priority queue implementations and various sorting algorithms like heap sort.

In summary, complete binary trees are a fundamental data structure with a well-defined structure that ensures efficient storage and navigation. They are widely used in computer science and have practical applications in areas such as data structures, algorithms, and memory management.

- **Algorithm:**

```
Function inorder(tree, index, n)
   If index < n AND tree[index] is not -1 Then
      inorder(tree, 2 * index + 1, n)
      Print tree[index]
      inorder(tree, 2 * index + 2, n)
Function preorder(tree, index, n)
   If index < n AND tree[index] is not -1 Then
      Print tree[index]
      preorder(tree, 2 * index + 1, n)
      preorder(tree, 2 * index + 2, n)
Function postorder(tree, index, n)
   If index < n AND tree[index] is not -1 Then
      postorder(tree, 2 * index + 1, n)
      postorder(tree, 2 * index + 2, n)
      Print tree[index]
Function levelOrder(tree, n)
   For i from 0 to n - 1
      If tree[i] is not -1 Then
         Print tree[i]
Function main()
   Declare n as Integer
   Print "Enter the number of elements: "
   Read n
   Declare tree as an Array of Integer with size n
   Initialize all elements of tree to -1
   For i from 0 to n - 1
      Print "Enter element ", i + 1, ": "
      Read tree[i]
   Declare choice as Integer
   Repeat
      Print "Menu:"
      Print "1. Inorder Traversal"
      Print "2. Preorder Traversal"
      Print "3. Postorder Traversal"
      Print "4. Level order Traversal"
      Print "5. Exit"
      Print "Enter your choice: "
      Read choice
      Switch choice
         Case 1:
            Print "Inorder Traversal: "
            Call inorder(tree, 0, n)
            Print newline
         Case 2:
            Print "Preorder Traversal: "
            Call preorder(tree, 0, n)
            Print newline
         Case 3:
            Print "Postorder Traversal: "
            Call postorder(tree, 0, n)
            Print newline
         Case 4:
            Print "Level Order Traversal: "
            Call levelOrder(tree, n)
            Print newline
         Case 5:
            Print "Exiting..."
         Default:
```

```
        Print "Invalid choice!"
    Until choice is 5
    Free memory allocated for tree
    Return 0
```

- **Source Code:**

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

void inorder(int tree[], int index, int n) {
    if (index < n && tree[index] != -1) {
        inorder(tree, 2 * index + 1, n);
        printf("%d ", tree[index]);
        inorder(tree, 2 * index + 2, n);
    }
}
void preorder(int tree[], int index, int n) {
    if (index < n && tree[index] != -1) {
        printf("%d ", tree[index]);
        preorder(tree, 2 * index + 1, n);
        preorder(tree, 2 * index + 2, n);
    }
}
void postorder(int tree[], int index, int n) {
    if (index < n && tree[index] != -1) {
        postorder(tree, 2 * index + 1, n);
        postorder(tree, 2 * index + 2, n);
        printf("%d ", tree[index]);
    }
}
void levelOrder(int tree[], int n) {
    for (int i = 0; i < n; i++) {
        if (tree[i] != -1) {
            printf("%d ", tree[i]);
        }
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int *tree = (int *)malloc(n * sizeof(int));
    memset(tree, -1, n*sizeof(tree[0]));
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &tree[i]);
    }
    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Inorder Traversal\n");
        printf("2. Preorder Traversal\n");
        printf("3. Postorder Traversal\n");
        printf("4. Level order Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
```

```
            printf("Inorder Traversal: ");
            inorder(tree, 0, n);
            printf("\n");
            break;
        case 2:
            printf("Preorder Traversal: ");
            preorder(tree, 0, n);
            printf("\n");
            break;
        case 3:
            printf("Postorder Traversal: ");
            postorder(tree, 0, n);
            printf("\n");
            break;
        case 4:
            printf("Level Order Traversal: ");
            levelOrder(tree, n);
            printf("\n");
            break;
        case 5:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice!\n");
    }
  } while (choice != 5);
  free(tree);
  return 0;
}
```

- **Explanation of the program:**

This program implements different tree traversal algorithms (inorder, preorder, postorder, and level order) for a complete binary tree. The program allows the user to input the elements of the binary tree and then choose which traversal algorithm to apply.
Here is a step-by-step explanation of the code:

1. There are four traversal functions defined in the code:
    - `inorder`: Performs an inorder traversal of the binary tree.
    - `preorder`: Performs a preorder traversal of the binary tree.
    - `postorder`: Performs a postorder traversal of the binary tree.
    - `levelOrder`: Performs a level order traversal of the binary tree.
2. In the `main` function:
    - It prompts the user to enter the number of elements in the binary tree and allocates memory for an integer array `tree` to store these elements.
    - It initializes all elements of the `tree` array to -1 using `memset`. This is a common practice to indicate that a node in the tree is empty or does not exist.
    - It then takes input from the user for the elements of the binary tree and stores them in the `tree` array.
    - It presents a menu to the user with the following options:
        - Inorder Traversal
        - Preorder Traversal
        - Postorder Traversal
        - Level Order Traversal
        - Exit
    - It uses a `do-while` loop to repeatedly display the menu and execute the selected traversal operation based on the user's choice.
    - Inside the loop, a `switch` statement is used to determine which traversal to perform based on the user's choice.
    - The selected traversal function is called with the `tree` array and the appropriate arguments to start the traversal from the root (index 0).

- After each traversal, the program prints the result.
- The loop continues until the user selects the "Exit" option (choice 5).
- Finally, the allocated memory for the `tree` array is freed, and the program terminates.

- **Expected Results:**

```
Enter the number of elements: 7
Enter element 1: 4
Enter element 2: 2
Enter element 3: 6
Enter element 4: 1
Enter element 5: 3
Enter element 6: 5
Enter element 7: 7

Menu:
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level order Traversal
5. Exit
Enter your choice: 1
Inorder Traversal: 1 2 3 4 5 6 7

Menu:
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level order Traversal
5. Exit
Enter your choice: 2
Preorder Traversal: 4 2 1 3 6 5 7

Menu:
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level order Traversal
5. Exit
Enter your choice: 3
Postorder Traversal: 1 3 2 5 7 6 4

Menu:
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level order Traversal
5. Exit
Enter your choice: 4
Level Order Traversal: 4 2 6 1 3 5 7

Menu:
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level order Traversal
5. Exit
Enter your choice: 5
Exiting...
```

16. Session – 11: Lab Experiment - Binary Tree Routing Table:

- **Problem Statement:**

Create a C program to demonstrate the use of binary search tree (BST) in implementing a basic routing table for a network. Binary search trees are a type of binary tree where each node has at most two children, and the values in the left subtree are smaller than the node's value, while the values in the right subtree are greater. This property allows for efficient searching and insertion. The program should output appropriate nextHop or "Route not found" message based on destination IP address.
The program should do the following tasks:
Binary Search Tree (BST): In a binary search tree routes must be organized based on the destination IP addresses. The left subtree of a node contains routes with smaller destination IP addresses, and the right subtree contains routes with larger destination IP addresses.
Route Insertion Criteria: Create a function insert to insert a new route into the binary search tree.

- If the tree is empty (i.e., root is NULL), a new node is created with the route and returned.
- If the tree is not empty, the function compares the destination IP address of the new route with the destination IP address of the current node.
- If the new route's destination IP is less than the current node's destination IP, the insertion continues in the left subtree recursively.
- If the new route's destination IP is greater than or equal to the current node's destination IP, the insertion continues in the right subtree recursively.
- The function returns the modified subtree after inserting the new route.

This structure ensures that when performing a routing lookup, the program can efficiently traverse the tree to find the appropriate next hop based on the destination IP address.
Look up: Create lookup function for performing a routing lookup in the binary search tree (BST) to find the appropriate next hop for a given destination IP address.
Delete Node: Create a function **deleteNode** to delete a node with a specific destination IP address from the binary tree routing table.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

1. analyze the efficiency and advantages of implementing a Binary Search Tree (BST) as a routing table in C, highlighting the trade-offs involved in comparison to other data structures.
2. create a functional Binary Search Tree-based routing table in C. They will implement key operations such as adding and deleting routes, as well as searching for routes within the BST.

- **Theory:**

A theoretical implication of using a Binary Search Tree (BST) as a routing table in a computer network is improved routing efficiency. Here's how this implication works:
In computer networking, a routing table is used to determine the optimal path for forwarding data packets from a source to a destination. Traditional routing tables often use linear data structures like lists or arrays, which can result in slower lookups and less efficient routing decisions, especially as the size of the routing table grows.
By employing a BST as the underlying data structure for the routing table, several advantages can be realized:

1. **Faster Lookup**: BSTs are inherently designed for efficient searching. When a router needs to determine the next hop for a specific destination IP address, it can perform a binary search in the BST. This means that the lookup time is proportional to the logarithm of the number of entries in the routing table, resulting in quicker routing decisions compared to linear data structures.
2. **Balanced Tree**: To maximize efficiency, the BST can be maintained in a balanced state, ensuring that the depth of the tree remains relatively small. This balance ensures that worst-case lookup times are minimized and that routing decisions are consistently fast.
3. **Ordered Routing Entries**: In a BST, entries are naturally ordered based on their IP addresses. This ordering simplifies tasks such as finding the longest prefix match, a critical operation in routing. The longest prefix match ensures that the router selects the most specific route when multiple routes match a destination IP address.
4. **Efficient Updates**: BSTs allow for efficient insertions and deletions. When network configurations change, routers can easily update the routing table without significant overhead. This feature is crucial in dynamic network environments where routes may change frequently.

5. **Reduced Memory Usage**: Compared to other data structures like hash tables, BSTs can use less memory while still providing efficient routing. This can be important in situations where memory resources are constrained.
6. **Scalability**: As the network grows and the routing table becomes larger, the BST-based routing table continues to provide efficient lookups. This scalability is vital for large-scale networks and internet routing.

However, it's important to note that while BSTs offer many advantages for routing tables, they are not without limitations. They may not be the best choice for scenarios with extremely fast insertions and deletions or highly dynamic routing tables, where other data structures like trie-based tables or hash tables might be more suitable. The choice of a data structure for routing tables should consider the specific requirements and characteristics of the network in question.

- **Algorithm - Binary Search Tree Routing Table:**

Data Structures:
- Route: Structure with destIP and nextHop fields
- Node: Structure with route, left, and right pointers

Functions:
1. createNode(route): Create a new Node with the given route.
2. insert(root, route): Insert a route into the BST.
3. findMin(node): Find the minimum node in the BST.
4. deleteNode(root, destIP): Delete a node with the given destination IP from the BST.
5. lookup(root, destIP): Look up the next hop for a given destination IP.
6. freeTree(root): Free the memory used by the BST.

Main Algorithm:
1. Initialize the root node as NULL.
2. Insert routes into the routing table using the insert function:
    - For each route to be added:
        - Create a Node using createNode with the route.
        - Call the insert function to add the Node to the BST.
3. Lookup the next hop for a destination IP using the lookup function:
    - For each destination IP to be looked up:
        - Call the lookup function with the root node and the destination IP.
        - Print the result.
4. Delete a route from the routing table using the deleteNode function:
    - For each route to be deleted:
        - Call the deleteNode function with the root node and the destination IP.
        - Print a message indicating the route was deleted.
5. Free the memory used by the routing table using the freeTree function:
    - Call the freeTree function with the root node.

- **Source Code:**

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

typedef struct Route {
   char destIP[20];
   char nextHop[20];
} Route;

typedef struct Node {
   Route route;
   struct Node* left;
   struct Node* right;
} Node;

Node* createNode(Route route) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   if (newNode) {
      newNode->route = route;
      newNode->left = NULL;
      newNode->right = NULL;
```

```
    }
    return newNode;
}
Node* insert(Node* root, Route route) {
    if (root == NULL)
        return createNode(route);
    if (strcmp(route.destIP, root->route.destIP) < 0)
        root->left = insert(root->left, route);
    else
        root->right = insert(root->right, route);
    return root;
}
Node* findMin(Node* node) {
    if (node == NULL)
        return NULL;
    while (node->left != NULL)
        node = node->left;
    return node;
}
Node* deleteNode(Node* root, char destIP[]) {
    if (root == NULL)
        return root;
    if (strcmp(destIP, root->route.destIP) < 0)
        root->left = deleteNode(root->left, destIP);
    else if (strcmp(destIP, root->route.destIP) > 0)
        root->right = deleteNode(root->right, destIP);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = findMin(root->right);
        root->route = temp->route;
        root->right = deleteNode(root->right, temp->route.destIP);
    }
    return root;
}
char* lookup(Node* root, char destIP[]) {
    if (root == NULL)
        return "Route not found";
    if (strcmp(destIP, root->route.destIP) == 0)
        return root->route.nextHop;
    else if (strcmp(destIP, root->route.destIP) < 0)
        return lookup(root->left, destIP);
    else
        return lookup(root->right, destIP);
}
void freeTree(Node* root) {
    if (root == NULL)
        return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}
```

```
int main() {
    Node* routingTable = NULL;

    Route route1 = {"192.168.1.0", "Gateway A"};
    Route route2 = {"10.0.0.0", "Gateway B"};
    Route route3 = {"172.16.0.0", "Gateway C"};

    routingTable = insert(routingTable, route1);
    routingTable = insert(routingTable, route2);
    routingTable = insert(routingTable, route3);

    char destIP[20];
    strcpy(destIP, "10.0.0.0");
    printf("Destination IP: %s, Next Hop: %s\n", destIP, lookup(routingTable, destIP));

    strcpy(destIP, "192.168.1.0");
    printf("Destination IP: %s, Next Hop: %s\n", destIP, lookup(routingTable, destIP));

    strcpy(destIP, "172.16.0.0");
    printf("Destination IP: %s, Next Hop: %s\n", destIP, lookup(routingTable, destIP));

    // Delete a route
    strcpy(destIP, "10.0.0.0");
    routingTable = deleteNode(routingTable, destIP);

    printf("Route deleted: %s\n", destIP);
    strcpy(destIP, "10.0.0.0");
    printf("Destination IP: %s, Next Hop: %s\n", destIP, lookup(routingTable, destIP));

    freeTree(routingTable);

    return 0;
}
```

- **Explanation of the program:**

This program implements a basic Binary Search Tree (BST) to create, modify, and query a routing table for network routing. Let's break down the program step by step:

1. **Structs:**
   - Two custom data structures are defined:
     - `Route`: Represents a network route with destination IP address (`destIP`) and the next hop (`nextHop`) for that route.
     - `Node`: Represents a node in the Binary Search Tree. Each node contains a `Route` struct and pointers to its left and right children.
2. **`createNode` Function:**
   - `createNode` is a utility function that allocates memory for a new `Node`, initializes it with the provided `Route`, and sets its children pointers to `NULL`. It returns a pointer to the newly created node.
3. **`insert` Function:**
   - `insert` adds a new `Route` to the BST.
   - It takes the current root of the BST (`root`) and the `Route` to be inserted (`route`) as parameters.
   - If the tree is empty (root is `NULL`), it creates a new node and returns it as the new root.
   - Otherwise, it compares the `destIP` of the `route` with the `destIP` of the current node to determine whether to insert the new route in the left or right subtree accordingly.
   - Recursion is used to insert the route in the appropriate subtree.
4. **`findMin` Function:**
   - `findMin` is a utility function to find the node with the minimum value (leftmost node) in a BST.
   - It iterates through the left children of nodes until it reaches the leftmost node.
5. **`deleteNode` Function:**
   - `deleteNode` removes a route with a specific destination IP from the BST.

- It takes the current root of the BST (`root`) and the destination IP (`destIP`) to be deleted as parameters.
- The function first searches for the node containing the specified `destIP` by recursively traversing the tree.
- When the node is found, it is deleted based on the following cases:
  - If the node has no children or only one child, it is replaced by its child (if any).
  - If the node has two children, it is replaced by the minimum node in its right subtree, and the minimum node is deleted recursively.

6. **`lookup` Function:**
   - `lookup` searches for a route in the BST based on a given destination IP address (`destIP`).
   - It takes the current root of the BST (`root`) and the destination IP (`destIP`) to be looked up as parameters.
   - The function recursively compares `destIP` with the `destIP` of nodes to navigate the tree.
   - When the desired node is found, it returns the corresponding `nextHop`. If not found, it returns "Route not found."
7. **`freeTree` Function:**
   - `freeTree` recursively frees the memory allocated for all nodes in the BST. It starts from the root and traverses all nodes, freeing them one by one.
8. **`main` Function:**
   - In the `main` function:
     - It initializes an empty routing table (`routingTable`) represented as a BST.
     - Three routes are inserted into the routing table using the `insert` function.
     - The program performs a series of lookups and prints the corresponding next hops based on destination IPs.
     - It then deletes a route from the routing table using the `deleteNode` function and verifies the deletion.
     - Finally, the program frees the memory of the entire routing table using the `freeTree` function.

- **Expected Results:**

Destination IP: 10.0.0.0, Next Hop: Gateway B
Destination IP: 192.168.1.0, Next Hop: Gateway A
Destination IP: 172.16.0.0, Next Hop: Gateway C
Route deleted: 10.0.0.0
Destination IP: 10.0.0.0, Next Hop: Route not found.

17. Session – 12: Assignment - Implementing a Symbol Table using Binary Search Tree (BST):

- **Problem Statement:**

In this programming assignment, you will have the opportunity to implement a symbol table using a Binary Search Tree (BST). A symbol table is a fundamental data structure used in various applications to store key-value pairs. In this case, you will be creating a symbol table that stores words and their corresponding frequencies.

Your task is to implement a C program that demonstrates the use of a Binary Search Tree as a symbol table. The program should allow users to input words, which will be inserted into the BST. Each word's frequency will be tracked. After inserting words, the program should display the contents of the symbol table and prompt the user to enter a word for searching. If the word is found in the symbol table, its frequency should be displayed; otherwise, a message indicating that the word is not in the table should be shown.

- **Student Learning Outcomes:**

After successful execution of this program, the student shall be able to

- evaluate the structure and properties of Binary Search Trees (BSTs) to design an efficient symbol table implementation, identifying key characteristics that make BSTs suitable for symbol table operations.
- develop a functional symbol table using a Binary Search Tree (BST) data structure, implementing essential operations such as insertion, deletion, and retrieval, ensuring the symbol table adheres to the principles of a BST for efficient symbol management.

PART – B

DATA STRUCTURES PROJECT REPORT FORMAT

18. Project Report Format:

REVA UNIVERSITY
Bengaluru, India

School of Computing Science and Engineering

Course Code:	**Data Structure Project Report**	**Academic Year:**
		Semester & Batch:
Project Details:		
Project Title:		
Place of Project:	**REVA UNIVERSITY, BENGALURU**	
Student Details:		
Name:		**Sign:**
Mobile No:		
Email-ID:		
SRN:		
Guide and Lab Faculty Members Details		
Guide Name:		**Sign:** **Date:**
Grade by Guide:		
Name of Lab Faculty 1		**Sign:** **Date:**
Name of Lab Faculty 2		**Sign:** **Date:**
Grade by Lab Faculty Members (combined)		
SEE Examiners		
Name of Examiner 1:		**Sign:** **Date:**
Name of Examiner 2:		**Sign:** **Date:**

Contents

1. *Abstract:*

An abstract is an outline/brief summary of your paper and your whole project. It should have an intro, body and conclusion. It is a well-developed paragraph, should be exact in wording, and must be understandable to a wide audience. Abstracts highlight major points of your project and explain why your work is important; what your purpose was, how you went about your project, what you learned, and what you concluded.

Keywords: -
(Font size: 12
Fount name: Time new roman in Italic style
Line spacing: 1.15
Abstract should be between 150 to 250 words)

2. Introduction:

The introduction section of a project serves as the opening statement that sets the stage for the entire project. It provides readers with an overview of what to expect, why the project is important, and what motivated its initiation. In other words, the introduction sets the tone for the entire project, providing a roadmap for readers and justifying the project's existence. It should be well-structured, engaging, and informative, motivating readers to continue exploring the project report.

Introduction and Remaining of the document will follow the following format.
Font size: 12
Fount name: Time new roman
Line spacing: 1.15
Introduction should be between approximately ½ to 1 page.

3. Problem statement:

Write your problem statement here.

A problem statement is usually one or two sentences to explain the problem your project will address. In general, a problem statement will outline the negative points of the current situation and explain why these matters. It also serves as a great communication tool, helping to get buy-in and support from others. One of the most important goals of any problem statement is to define the problem being addressed in a way that's clear and precise.

4. Project overview:

Provide the project overview according to the components mentioned here.

A project overview is a detailed description of a project's goals and objectives, the steps to achieve these goals, and the expected outcomes.

4.1. **Objectives:** An objective describes the desired results of a project, which often includes a tangible item. An objective is specific and measurable, and must meet time, budget, and quality constraints. A project may have one objective, many parallel objectives, or several objectives that must be achieved sequentially. To produce the most benefit, objectives must be defined early in the project life cycle, in phase one, the planning phase.

4.2. **Goals:** The goal of a project overview is to lay out the details of a project in a concise, easy-to-understand manner that can be presented to clients, team members, and key stakeholders.

5. Project Implementation

Provide the analysis of project, the functions identified to be implemented and finally list the complete commented source code.

Your project group is required to submit a document outlining the project's implementation details. Ensure that your code follows proper coding conventions. Include appropriate comments on critical sections of the code. Overall, your code should have a smooth flow, logical transitions, and should be easy to follow.

5.1. Problem analysis and description.

The problem analysis and description section is a critical component that outlines the specific issue or challenge that the project aims to address. It serves as the foundation upon which the rest of the project is built and helps stakeholders, team members, and anyone else involved or interested in the project to understand the problem and its context.

5.2. Modules identified:

A "module" is a high-level description of a functional area, consisting of a group of processes describing the functionality of the module and a group of packages implementing the functionality.

5.3. Code with comments.

A "code with comments" refers to a piece of computer programming code that includes explanatory comments written alongside the actual code. These comments are meant to provide additional information, explanations, and context about what the code does, how it works, and why certain decisions were made during the coding process. The primary purpose of code comments is to make the code more understandable and maintainable for both the original developer and others who may need to read or modify it in the future.

6. Output and results

Attach the output generated by your project and results. (Screenshot of output and description for results or impact of project)

Outputs" and "results" are two distinct but interconnected aspects of a project, and they are often used to measure the project's success and impact. In short, outputs are the tangible products and deliverables produced during a project's execution, while results are the broader and often longer-term changes or benefits that occur as a direct or indirect consequence of these outputs. Both output and results are essential for evaluating a project's success and effectiveness, with results being the ultimate measure of the project's impact on its intended beneficiaries or stakeholders.

7. Conclusions:

Write the conclusion here.

A conclusion is the last part of something, it means "finally, to sum up," and is used to introduce some final comments at the end of writing.

8. References:

References in a document are citations or sources of information that support and substantiate the content presented in the document. Properly formatted references enhance the credibility and integrity of your writing while giving readers the means to verify and explore the sources you've used.

It's essential to follow the specific citation style guidelines for formatting your references correctly. Additionally, make sure your in-text citations (citations within the main body of your document) correspond to the entries in your references section.

References must be quoted as per the following format:

Author(s) Initial(s). Surname(s), "Title of Report," Abbrev. Name of Co., City of Co., Abbrev. State, Country (abbrev. US State or Country if the City is not 'well known'), Report number/Type (if available), Abbrev. Month. (Day if available), Year of Publication.

Example for reference is given below. Kindly follow the same format for writing reference

G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

K. Elissa, "Title of paper if known," unpublished.

R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

19. Learning Resources:

Books:

1. "Data Structures Using C" by Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein
2. "Data Structures and Algorithms in C" by Michael T. Goodrich, Roberto Tamassia, and David M. Mount
3. "Algorithms in C, Parts 1-5" by Robert Sedgewick
4. "Data Structures Through C in Depth" by S. K. Srivastava and Deepali Srivastava
5. "C Programming Absolute Beginner's Guide" by Perry and Miller
6. "Data Structures and Algorithm Analysis in C" by Mark A. Weiss
7. "Data Structures and Program Design in C" by Robert L. Kruse, Alexander J. Ryba, and Bruce P. Leung
8. "C & Data Structures" by P.S. Deshpande and O.G. Kakde
9. "Data Structures: A Pseudocode Approach with C" by Richard F. Gilberg and Behrouz A. Forouzan

Online Courses and Tutorials:

1. Coursera - "Data Structures" (offered by University of California, San Diego)
2. edX - "Advanced Data Structures in C" (offered by University of California, Irvine)
3. GeeksforGeeks (geeksforgeeks.org) - Offers a wide range of tutorials and articles on data structures in C.
4. Khan Academy - "Algorithms" (khanacademy.org) - Covers basic algorithms and data structures.
5. Udemy - "Data Structures and Algorithms: Deep Dive Using C" by Tim Buchalka

Websites and Online Resources:

1. CodeChef (codechef.com) - Provides practice problems and contests to improve your data structure skills in C.
2. HackerRank (hackerrank.com) - Offers coding challenges and data structure problems in C.
3. LeetCode (leetcode.com) - Features a vast collection of coding problems, including data structure challenges in C.
4. Stack Overflow (stackoverflow.com) - A valuable resource for finding answers to specific questions related to data structures in C.
5. GitHub (github.com) - Explore open-source C projects that implement various data structures for hands-on learning.