Dayananda Sagar University

School of Engineering

Department of Computer Science and Engineering

Data Structure

Laboratory Manual

Department of Computer Science and Engineering

Data Structure

Name:	
University Serial Number:	

Vision and Mission of the University

Vision
To be a Centre of excellence in education, research & training, innovation & entrepreneurship and to produce citizens with exceptional leadership qualities to serve national and global needs.

Mission
To achieve our objectives in an environment that enhances creativity, innovation and scholarly pursuits while adhering to our vision.

Values
• The values that drive DSU and support its vision: The Pursuit of Excellence
• A commitment to strive continuously to improve ourselves and our systems with the aim of becoming the best in our field.

Fairness
A commitment to objectivity and impartiality, to earn the trust and respect of society.

Leadership
A commitment to lead responsively and creatively in educational and research processes.

Integrity and Transparency
A commitment to be ethical, sincere and transparent in all activities and to treat all individuals with dignity and respect.

Vision and Mission of the Department

Vision

To develop pool of high calibre professionals, researchers and entrepreneurs in the areas of Computer Science & Engineering and Information Technology with exceptional technical expertise, skills and ethical values, capable of providing innovative solutions to the national and global needs.

Mission

- To create a robust ecosystem where academicians, concept developers, product designers, business incubators, product developers, entrepreneurs, mentors and financial institutions are brought together under one platform of the department.

- To establish Project Environment in the Department with open source tools, provide hands-on experience to students by establishing a process to channelize their effort towards acquiring relevant competencies and skills in their chosen technology areas and domains.

- To create continuous learning environment for faculty and establish Research Centres in collaboration with Industries and Institutions of National/International repute and conduct research in emerging areas as well as socially relevant technical and domain areas through funded research projects.

Program Educational Objectives (PEO's)

PEO1: Engage in the design, development, testing/verification and validation, and operation of computational systems in the field of Information Technology and related areas, or in multidisciplinary teams in any field where computing can be applied.

PEO2: Solve problems of social relevance applying the knowledge of Computer Science Engineering and/or pursue higher education and
research.

PEO3: Work effectively as professional and as team members in computing in multidisciplinary projects, and demonstrating initiative, persistence in problem solving, and excellent technical communication skills.

PEO4: Engage in lifelong, self-directed learning and career enhancement, anticipate changing professional and societal needs, and adapt rapidly to these changing needs.

Programme Outcome (PO's)

PO1: Engineering knowledge: Apply the knowledge of mathematics,
science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research- based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSO's)

PSO1: Develop, Analyse, Review and Contribute to efficient, secure and high quality design, implementation, testing and operations of computing system

PSO2: Find and articulate digital and intelligent solution that can fully or partially automate various aspects of human activity.

Dayananda Sagar University

SCHOOL OF ENGINEERING

Laboratory Certificate

This is to certify that Mr./Ms ____bearing University Seat number (USN)________________________has satisfactorily completed the experiments in Data Structure Laboratory prescribed by the University for the 3rd semester B.Tech program during the year 2023-2024.

Date:___________________

Marks	
Maximum	Obtained

Signature of the Faculty In charge

Signature of the Chairman

Laboratory Environment

- This course will use Ubuntu distribution of Linux as the Operating System for all the exercises
- The tool chain used will be GNU ‘gcc’ suite of tools
- Editor recommended is ‘gedit’ which is available in these systems
- ‘terminal’ program is used to run ‘gcc’ and the executable code

Basic Instructions to use the tools

1. If the system is not powered on, please turn the system ON by pressing the power button on the PC.
2. Check if the monitor needs to be turned on as well
3. If presented with a boot time option, please select ‘Ubuntu’ in the list
4. Once Ubuntu has booted up, click on the program selection icon on the bottom left of the screen (a
1. small square with dots).
5. Type ‘terminal’ in the search bar at the top
6. The icon for the ‘terminal’ program will appear below, and click the same to start the terminal
2. program
7. The terminal program will typically be a black coloured window with a prompt to type commands.
8. Type ‘gedit <prog_name.c>’ on the terminal window.
9. This will open the gedit window, which can be used to type the program. Note that the name of the
3. file you specified (prog_name.c) will appear on the top of the window. If there is a ‘*’ character before the file name, it indicates that the file has not been saved yet. Please remember to save the file once you have finished typing the program.
10. You can close the ‘gedit’ window to get back to the terminal prompt.
11. Type ‘gcc <prog_name.c>’ and press enter.
12. If there are no errors/warnings, you will see the command prompt again. Else, ‘gcc’ will print the errors and warnings with line numbers to help you fix the errors.
13. If there are errors/warnings, fix them and re-run ‘gcc’ till you get a clean compile.
14. ‘gcc’ by default will save the executable code in a file by name ‘a.out’.
15. To run your program and check the output, please type ‘./a.out’ and press enter. This will make the
4. OS run the code and you can check the output.
16. If the code does not give the desired output, or fails with some errors, repeat the edit-compile-test cycle by going back and editing the program using ‘gedit’.
    For example, if you want to name the file as lab1.c, the following are the commands
    ```
    gedit lab1.c
    gcc lab1.c
    ./a.out
    ```

Department of Computer Science and Engineering Academic Year 2023-2024/ 3rd Semester

Data Structure Laboratory

Index

Date:__ / __ / ____

Experiment 1

1. a. Write a C program to implement an array of fixed dimension. Perform the following operations inserting an element into an array and display the contents of an array. Note: user input is required to enter the size of an array.

Program

```
#include <stdio.h>
int main()
{
   int n;
   printf("Enter the size of the array:");
   scanf("%d",&n);
   int arr[n];

   printf("Enter the %d values to store it in array: \n", n);
   for(int i=0;i<n;i++)
   {
      scanf("%d",&arr[i]);
   }
   printf("The values stored in the array are: \n");
   for(int i=0;i<n;i++)
   {
      printf("%d \n",arr[i]);
   }

}
```

Sample Output

```
Enter the size of the array: 4
Enter the 4 values to store it in array: 1
2
3
4
The values stored in the array are:
1
2
3
4
```

1. b. Evaluate the Number Plates of Vehicles.

Write a C program to insert a vehicle registration number into an array of 1x10 array. The objective is to apply strict conventional rules of vehicle number plate registration as follows: The first two positions of array must be filled with state name, next two positions must be filled with district code, followed by next two positions with serial number of characters and penultimate with a number of a vehicle. KA-09-MN-3865. The program needs to verify the registration of a vehicle in the same format and print the registration of a vehicle and “Accept” else print “Reject” with a proper message.

Rules to check.

i. First two positions of an array must be a State Code.
ii. Next two positions of an array must be District Code.
iii. Next two Positions of an array must be a Serial Number of an RTO.
iv. Penultimate r positions of an array must be a vehicle number.

Program

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
// Function to verify if a vehicle registration number is valid
bool verifyRegistration(char registration[])
{
   // Check if the registration number has exactly 10 characters
   if (strlen(registration) != 10)
   {
      return false;
   }

   // Check if the first two characters are alphabets (e.g., "KA" for Karnataka)
   for (int i = 0; i < 2; i++)
   {
      if (!isalpha(registration[i]))
      {
         return false;
      }
   }

   // Check if the next two characters are numbers (e.g., "09" for a specific district)
   for (int i = 2; i < 4; i++)
   {
      if (!isdigit(registration[i]))
      {
         return false;
      }
   }

   // Check if the next two characters are alphabets (e.g., "MN" for serial number)
   for (int i = 4; i < 6; i++)
```

```
    {
        if (!isalpha(registration[i]))
        {
            return false;
        }
    }

    // Check if the penultimate two characters are numbers (e.g., "3865" for vehicle number)
    for (int i = 6; i < 10; i++)
    {
        if (!isdigit(registration[i]))
        {
            return false;
        }
    }

    return true;
}

int main() {
    char registration[11];
    printf("Enter a vehicle registration number: ");
    scanf("%s", registration);

    if (verifyRegistration(registration))
    {
        printf("Accept: Vehicle registration number is valid.\n");
    }
    else
    {
        printf("Reject: Vehicle registration number is invalid.\n");
    }

    return 0;
}
```

Sample Output

```
Enter a vehicle registration number: KA09MN3865
Accept: Vehicle registration number is valid.

Enter a vehicle registration number: KAABMN3865
Reject: Vehicle registration number is invalid.
```

Date:__ / __ / ____

Experiment 2

2. a. Write a C program that multiplies two matrices, ensuring that the number of columns in the first matrix is equal to the number of rows in the second matrix. Display the resulting matrix. Note: Prompt user to enter the size of the matrix.

Program

```
#include <stdio.h>
int main() {
    int m, n, p, q;

    printf("Enter the number of rows and columns of the first matrix: ");
    scanf("%d %d", &m, &n);

    printf("Enter the number of rows and columns of the second matrix: ");
    scanf("%d %d", &p, &q);

    if (n != p) {
        printf("Matrix multiplication is not possible. Column of the first matrix must be equal to
the row of the second matrix.\n");
        return 1;
    }

    int firstMatrix[m][n], secondMatrix[p][q], resultMatrix[m][q];

    printf("Enter elements of the first matrix:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &firstMatrix[i][j]);
        }
    }

    printf("Enter elements of the second matrix:\n");
    for (int i = 0; i < p; i++) {
        for (int j = 0; j < q; j++) {
            scanf("%d", &secondMatrix[i][j]);
        }
    }

    // Matrix multiplication
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < q; j++) {
            resultMatrix[i][j] = 0;
            for (int k = 0; k < n; k++) {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }

    printf("Resultant matrix after multiplication:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < q; j++) {
            printf("%d ", resultMatrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

2. b. Validate the sudoku board.
Write a C program to determine the validity of a 9x9 Sudoku board. The task was to apply strict Sudoku rules to only the filled cells. With lines of code, scan the rows, columns, and 3x3 sub grids for reptations. The program verifies the Sudoku board's integrity and print "**Accept**" if it satisfies all the sudoku board condition, else print "**Reject**".

Conditions to check
i. Every row should encompass the numbers 1 through 9, ensuring no duplication.
ii. Each column should consist of the numbers 1 to 9, ensuring no duplication.
iii. Within the nine 3x3 sub-boxes of the grid, no repetitions allowed.

Sample Input
```
//copy and paste the input
int sudoku[9][9] = {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };
```

Sample Output
```
Accept
```

```
#include <stdio.h>
int isValidSudoku(int board[9][9]) {
    // Check rows
    for (int i = 0; i < 9; i++) {
        int row[10] = {0};
        for (int j = 0; j < 9; j++) {
            if (board[i][j] != 0 && row[board[i][j]] == 1) {
                return 0; // Invalid Sudoku
            }
            row[board[i][j]] = 1;
        }
    }

    // Check columns
    for (int j = 0; j < 9; j++) {
        int col[10] = {0};
        for (int i = 0; i < 9; i++) {
            if (board[i][j] != 0 && col[board[i][j]] == 1) {
                return 0; // Invalid Sudoku
            }
            col[board[i][j]] = 1;
        }
    }

    // Check 3x3 subgrids
    for (int block = 0; block < 9; block++) {
        int subgrid[10] = {0};
        for (int i = block / 3 * 3; i < block / 3 * 3 + 3; i++) {
            for (int j = block % 3 * 3; j < block % 3 * 3 + 3; j++) {
                if (board[i][j] != 0 && subgrid[board[i][j]] == 1) {
                    return 0; // Invalid Sudoku
                }
                subgrid[board[i][j]] = 1;
            }
        }
    }

    return 1; // Valid Sudoku
}

int main() {
    int sudoku[9][9] = {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };

    if (isValidSudoku(sudoku)) {
        printf("Valid Sudoku\n");
    } else {
        printf("Invalid Sudoku\n");
    }

    return 0;
}
```

Date:__ / __ / ____

Experiment 3

3. a. A school wants to develop a student academic system to track student progress and generate reports for teachers and parents. The system should store the following information for each student: Student ID, Name, Grade, Subject, Marks. The system should also be able to do the following operations:
i. Calculate the average marks for each student.
ii. Assign grades based on the average marks.

Conditions
i. Must use array of structures.
ii. Use simple structure variable to access the members of the structure.
ii. Enter marks for minimum of 5 students with subjects each.

Program
```
#include <stdio.h>

// Define the structure for a student
struct Student
{
   int studentID;
   char name[50];
   char grade;
   float marks[5];
   float averageMarks;
};

// Function to calculate the average marks for a student
void calculateAverage(struct Student *student)
{
   float totalMarks = 0.0;
   for (int i = 0; i < 5; i++)
   {
      totalMarks += student->marks[i];
   }
   student->averageMarks = totalMarks / 5;
}

// Function to assign grades based on average marks
void assignGrades(struct Student *student)
{
   if (student->averageMarks >= 90)
   {
      student->grade = 'A';
   }
   else if (student->averageMarks >= 80)
   {
      student->grade = 'B';
   }
   else if (student->averageMarks >= 70)
   {
      student->grade = 'C';
   }
   else if (student->averageMarks >= 60)
   {
      student->grade = 'D';
   }
   else
   {
      student->grade = 'F';
   }
}

int main()
{
   struct Student students[5];

   // Input student information
   for (int i = 0; i < 5; i++)
   {
      printf("Enter Student ID: ");
      scanf("%d", &students[i].studentID);
      printf("Enter Name: ");
      scanf("%s", students[i].name);

      printf("Enter marks for 5 subjects:\n");
      for (int j = 0; j < 5; j++) {
          printf("Subject %d: ", j + 1);
          scanf("%f", &students[i].marks[j]);
      }

      // Calculate average marks and assign grades
      calculateAverage(&students[i]);
      assignGrades(&students[i]);
   }

   // Display student information
   printf("\nStudent Information:\n");
   for (int i = 0; i < 5; i++)
   {
      printf("Student ID: %d\n", students[i].studentID);
      printf("Name: %s\n", students[i].name);
      printf("Average Marks: %.2f\n", students[i].averageMarks);
      printf("Grade: %c\n", students[i].grade);
      printf("\n");
   }

   return 0;
}
```

3. b. A school wants to develop a student academic system to track student progress and generate reports for teachers and parents. The system should store the following information for each student: Student ID, Name, Grade, Subject, Marks. The system should also be able to do the following operations:
i. Calculate the average marks for each student.
ii. Assign grades based on the average marks.

Conditions
i. Must use array of structures.
ii. Use **pointer to structure** to access the members of the structure.
ii. Enter marks for minimum of 5 students with subjects each.

Instructions to Faculties
If needed, please take extra lab hours in subsequent week to complete this experiment in detail. The concepts discussed in this experiment will make solid foundation for the forthcoming experiments namely (stack using linked list, queue using linked list, singly linked list, doubly linked list, circular linked list).

Program

```
#include <stdio.h>

// Define the structure for a student
struct Student
{
   int studentID;
   char name[50];
   char grade;
   float marks[5];
   float averageMarks;
};

// Function to calculate the average marks for a student
void calculateAverage(struct Student *student)
{
   float totalMarks = 0.0;
   for (int i = 0; i < 5; i++)
   {
      totalMarks += student->marks[i];
   }
   student->averageMarks = totalMarks / 5;
}

// Function to assign grades based on average marks
void assignGrades(struct Student *student)
{
   if (student->averageMarks >= 90)
```

```
    {
      student->grade = 'A';
    }
    else if (student->averageMarks >= 80)
    {
      student->grade = 'B';
    }
    else if (student->averageMarks >= 70)
    {
      student->grade = 'C';
    }
    else if (student->averageMarks >= 60)
    {
      student->grade = 'D';
    }
    else
    {
      student->grade = 'F';
    }
}

int main()
{
    struct Student students[5];
    struct Student *studentPtr = students;

    // Input student information
    for (int i = 0; i < 5; i++)
    {
      printf("Enter Student ID: ");
      scanf("%d", &studentPtr->studentID);
      printf("Enter Name: ");
      scanf("%s", studentPtr->name);

      printf("Enter marks for 5 subjects:\n");
      for (int j = 0; j < 5; j++)
      {
        printf("Subject %d: ", j + 1);
        scanf("%f", &studentPtr->marks[j]);
      }

      // Calculate average marks and assign grades
      calculateAverage(studentPtr);
      assignGrades(studentPtr);

      // Move to the next student in the array
      studentPtr++;
    }
    // Reset the pointer to the beginning of the array
    studentPtr = students;

    // Display student information
    printf("\nStudent Information:\n");
    for (int i = 0; i < 5; i++)
    {
        printf("Student ID: %d\n", studentPtr->studentID);
        printf("Name: %s\n", studentPtr->name);
        printf("Average Marks: %.2f\n", studentPtr->averageMarks);
        printf("Grade: %c\n", studentPtr->grade);
        printf("\n");

        // Move to the next student in the array
        studentPtr++;
    }

    return 0;
}
```

Date:__ / __ / ____

Experiment 4

4. a. Write a C program to create a stack data structure using a linked list instead of an array and implement push, pop and isEmpty operations accordingly.

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

// Define a structure for a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Define a structure for the stack
struct Stack {
    struct Node* top;
};

// Function to create an empty stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = NULL;
    return stack;
}

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

// Function to push an element onto the stack
void push(struct Stack* stack, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
    printf("%d pushed to the stack\n", data);
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. Cannot pop.\n");
        return -1; // Return an invalid value
    }
    struct Node* temp = stack->top;
    int poppedData = temp->data;
    stack->top = temp->next;
    free(temp);
    return poppedData;
}

int main() {
    struct Stack* stack = createStack();

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from the stack\n", pop(stack));
    printf("%d popped from the stack\n", pop(stack));
    printf("%d popped from the stack\n", pop(stack));
    printf("%d popped from the stack\n", pop(stack)); // Trying to pop from an empty stack

    return 0;
}
```

4. b. Design a web browser history tracker in C. Implement a stack data structure to keep track of visited URLs. Create functions to push new URLs onto the stack as users visit websites and pop URLs when users navigate backward in their browsing history.

Operations
i. Visit a URL (push)
ii. Go Back (pop)
iii. Display Current URL

i. Visit a URL: The program prompts the user to enter the URL (Uniform Resource Locator) they want to visit. The user enters the web address (e.g., "https://www.example.com") or a specific webpage URL. After the user provides the URL, the program creates a new node containing this URL and pushes it onto the stack. This action simulates the user's movement to a new web page. Additionally check if the stack overflow condition for insertion.

ii. Navigating Backward: In your C program for the web browser history tracker, ensure that users can navigate backward through their browsing history by popping URLs from the stack and displaying the current URL. Provide a mechanism for handling the case when there are no more URLs to go back to.

iii. Displaying Current URL: In your web browser history tracker program, create a function that displays the current URL that the user is on. Ensure that it correctly reflects the URL on the top of the stack.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_URL_LENGTH 100

// Structure for a node in the stack
typedef struct Node {
   char url[MAX_URL_LENGTH];
   struct Node* next;
} Node;

// Structure for the stack
typedef struct {
   Node* top;
} Stack;

// Function to initialize an empty stack
void initialize(Stack* stack) {
   stack->top = NULL;
}

// Function to push a URL onto the stack
```

```
void push(Stack* stack, const char* url) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    strncpy(newNode->url, url, MAX_URL_LENGTH);
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to pop a URL from the stack
void pop(Stack* stack) {
    if (stack->top == NULL) {
        printf("No more URLs in the history.\n");
        return;
    }

    Node* temp = stack->top;
    stack->top = stack->top->next;
    free(temp);
}

// Function to display the current URL
void displayCurrentURL(Stack* stack) {
    if (stack->top == NULL) {
        printf("No URL currently loaded.\n");
    } else {
        printf("Current URL: %s\n", stack->top->url);
    }
}

int main() {
    Stack historyStack;
    initialize(&historyStack);

    int choice;
    char url[MAX_URL_LENGTH];

    while (1) {
        printf("\nMenu:\n");
        printf("1. Visit a URL\n");
        printf("2. Go Back\n");
        printf("3. Display Current URL\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter URL to visit: ");
                scanf("%s", url);
                push(&historyStack, url);
                break;
            case 2:
                pop(&historyStack);
                break;
            case 3:
                displayCurrentURL(&historyStack);
                break;
            case 4:
                // Clean up and exit the program
                while (historyStack.top != NULL) {
                    pop(&historyStack);
                }
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Sample Output

```
Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 1
Enter URL to visit: example.com

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 1
Enter URL to visit: Meow

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
```

```
Enter your choice: 3
Current URL: Meow

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 2

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 3
Current URL: example.com

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 2

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 3
No URL currently loaded.

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 2
No more URLs in the history.

Menu:
1. Visit a URL
2. Go Back
3. Display Current URL
4. Exit
Enter your choice: 4
```

Date:__ / __ / ____

Experiment 5

5. a. Write a C program to implement a queue data structure using a singly linked list. Your program should provide the following functionalities:

1. Initialize an empty queue.
2. Enqueue (add) an element to the rear of the queue.
3. Dequeue (remove) an element from the front of the queue.
4. Display the elements of the queue.

You need to define suitable functions to perform these operations and demonstrate their usage in your program. Ensure that you handle edge cases such as queue underflow (when trying to dequeue from an empty queue).

Program

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

// Structure to represent a node in the linked list
struct Node
{
   int data;
   struct Node* next;
};

// Structure to represent a queue
struct Queue
{
   struct Node* front;
   struct Node* rear;
};

// Function to initialize an empty queue
struct Queue* createQueue()
{
   struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
   queue->front = queue->rear = NULL;
   return queue;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue)
{
   return (queue->front == NULL);
```

```
}

// Function to enqueue an element to the rear of the queue
void enqueue(struct Queue* queue, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;

   if (isEmpty(queue))
   {
      queue->front = queue->rear = newNode;
   } else
   {
      queue->rear->next = newNode;
      queue->rear = newNode;
   }
}

// Function to dequeue an element from the front of the queue
int dequeue(struct Queue* queue)
{
   if (isEmpty(queue))
   {
      printf("Queue underflow: Cannot dequeue from an empty queue.\n");
      return -1; // Error value
   }

   struct Node* temp = queue->front;
   int data = temp->data;
   queue->front = queue->front->next;

   free(temp);

   return data;
}

// Function to display the elements of the queue
void display(struct Queue* queue)
{
   if (isEmpty(queue))
   {
      printf("Queue is empty.\n");
      return;
   }

   struct Node* current = queue->front;
   printf("Queue elements: ");

   while (current != NULL)
```

```
        {
            printf("%d ", current->data);
            current = current->next;
        }

    printf("\n");
}

int main()
{
    struct Queue* queue = createQueue();

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);

    display(queue);

    printf("Dequeued element: %d\n", dequeue(queue));

    display(queue);

    printf("Is the queue empty? %s\n", isEmpty(queue) ? "Yes" : "No");

    return 0;
}
```

5. b. Imagine you are responsible for designing a queue-based system to manage the queue of regular customers waiting to purchase cinema tickets at a popular movie theatre. Your system should ensure fair and efficient ticket sales for all customers. When a customer's arrive at the cinema, they join the queue. Each customer is represented as name, age (for record-keeping), and number of tickets needed. When a customer reaches the front of the queue, they are served by the ticketing agent. Implement a ticket sale process where the agent provides the customer with the requested ticket(s). Initialize the total number of tickets and if the tickets are sold, then ticketing agent should display an houseful message.

Program

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

// Structure to represent a customer
struct Customer
{
   char name[50];
   int age;
   int numTickets;
   struct Customer* next;
};

// Structure to represent the queue
struct Queue
{
   struct Customer* front;
   struct Customer* rear;
};

// Initialize a queue
struct Queue* createQueue()
{
   struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
   queue->front = queue->rear = NULL;
   return queue;
}

// Check if the queue is empty
int isEmpty(struct Queue* queue)
{
   return (queue->front == NULL);
}

// Add a customer to the rear of the queue
void enqueue(struct Queue* queue, struct Customer customer)
{
   struct Customer* newNode = (struct Customer*)malloc(sizeof(struct Customer));
   *newNode = customer;
```

```
    newNode->next = NULL;

    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Remove and return a customer from the front of the queue
struct Customer dequeue(struct Queue* queue)
{
    struct Customer emptyCustomer = {"", 0, 0, NULL};

    if (isEmpty(queue))
    {
        return emptyCustomer; // Queue is empty
    }

    struct Customer* temp = queue->front;
    struct Customer customer = *temp;
    queue->front = queue->front->next;

    free(temp);
    return customer;
}


int main()
{
    // Total number of tickets available
    int totalTickets = 100;
    struct Queue* ticketQueue = createQueue();

    while (totalTickets > 0)
    {
        struct Customer customer;
        printf("Enter customer name: ");
        scanf("%s", customer.name);
        printf("Enter customer age: ");
        scanf("%d", &customer.age);
        printf("Enter number of tickets needed: ");
        scanf("%d", &customer.numTickets);

        if (customer.numTickets <= totalTickets)
        {
            totalTickets -= customer.numTickets;
            enqueue(ticketQueue, customer);
            printf("Tickets sold to %s (%d tickets remaining)\n", customer.name, totalTickets);
        }
        else
        {
            printf("Insufficient tickets available. Tickets not sold to %s\n", customer.name);
        }

        printf("Do you want to add another customer? (1 for yes, 0 for no): ");
        int choice;
        scanf("%d", &choice);
        if (choice != 1)
        {
            break;
        }
    }

    printf("Houseful! All tickets are sold.\n");

    return 0;
}
```

Date:__ / __ / ____

Experiment 6

6. a. Write a C program to implement a singly linked list. The program should allow the user to perform the following operations:

i. Insert a node at the beginning of the list.
ii. Insert a node at the end of the list.
iii. Delete a node by value.
iv. Display the linked list.

Additionally, ensure that you handle edge cases gracefully, such as an empty list or attempting to delete a node that does not exist.

Program

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

// Define a structure for a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a node at the beginning of the list
struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }

    newNode->data = value;
    newNode->next = head;
    return newNode;
}

// Function to insert a node at the end of the list
struct Node* insertAtEnd(struct Node* head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        return newNode;
    }

    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;

    return head;
}

// Function to delete a node by value
struct Node* deleteByValue(struct Node* head, int value) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return head;
    }

    if (head->data == value) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    struct Node* current = head;
    while (current->next != NULL && current->next->data != value) {
        current = current->next;
    }

    if (current->next == NULL) {
        printf("Value not found in the list. Cannot delete.\n");
        return head;
    }

    struct Node* temp = current->next;
    current->next = current->next->next;
    free(temp);

    return head;
}

// Function to display the linked list
void displayList(struct Node* head) {
    printf("Linked List: ");
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    int choice, value;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at the beginning\n");
        printf("2. Insert at the end\n");
        printf("3. Delete by value\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter a value to insert: ");
                scanf("%d", &value);
                head = insertAtBeginning(head, value);
                break;
            case 2:
                printf("Enter a value to insert: ");
                scanf("%d", &value);
                head = insertAtEnd(head, value);
                break;
            case 3:
                printf("Enter a value to delete: ");
                scanf("%d", &value);
                head = deleteByValue(head, value);
                break;
            case 4:
                displayList(head);
                break;
            case 5:
                // Clean up and exit
                while (head != NULL) {
                    struct Node* temp = head;
                    head = head->next;
                    free(temp);
                }
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

6. b. You are given a singly linked list where each node has a value and a reference (or pointer) to the next node in the list. For example, a linked list might look like this: 1 -> 2 -> 3 -> 4 -> 5, where each number represents the value in a node, and the arrow (->) represents the next pointer.

Task: Your task is to swap every two adjacent nodes in the list and return the new head of the modified list. In other words, you need to rearrange the nodes so that the linked list now looks like 2 -> 1 -> 4 -> 3 -> 5.

Constraints:
You are not allowed to modify the values in the nodes. Only the nodes themselves may be changed. If the number of nodes in the linked list is odd, the last node should remain in its original position.

Sample Output
Input: 1 -> 2 -> 3 -> 4 -> 5
Output: 2 -> 1 -> 4 -> 3 -> 5

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

// Define the structure for a node in the linked list
struct ListNode {
    int val;
    struct ListNode* next;
};

// Function to swap every two adjacent nodes in the linked list
struct ListNode* swapPairs(struct ListNode* head) {
    // Check if the list is empty or has only one node
    if (head == NULL || head->next == NULL) {
        return head;
    }

    // Initialize pointers for swapping
    struct ListNode* prev = NULL;
    struct ListNode* current = head;
    struct ListNode* nextNode = head->next;

    // Update the head to the second node
    head = nextNode;

    // Perform swaps in pairs
    while (current != NULL && nextNode != NULL) {
        // Adjust pointers for swapping
        current->next = nextNode->next;
        nextNode->next = current;
        if (prev != NULL) {
            // Connect the previous pair to the current pair
            prev->next = nextNode;
        }

        // Move to the next pair
        prev = current;
        current = current->next;

        if (current != NULL) {
            nextNode = current->next;
        }
    }

    return head;
}

// Function to print the linked list
void printList(struct ListNode* head) {
    struct ListNode* current = head;
    while (current != NULL) {
        printf("%d -> ", current->val);
        current = current->next;
    }
    printf("NULL\n");
}

// Function to create a new node with a given value
struct ListNode* newNode(int value) {
    struct ListNode* node = (struct ListNode*)malloc(sizeof(struct ListNode));
    if (node != NULL) {
        node->val = value;
        node->next = NULL;
    }
    return node;
}

int main() {
    // Create a sample linked list: 1 -> 2 -> 3 -> 4 -> 5
    struct ListNode* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    printf("Original Linked List: ");
    printList(head);
    // Swap adjacent nodes
    head = swapPairs(head);

    printf("Modified Linked List: ");
    printList(head);

    // Clean up memory
    while (head != NULL) {
        struct ListNode* temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```

Date:__ / __ / ____

Experiment 7

7. a. Write a C program to implement a doubly linked list data structure from scratch. The program should provide the following functionalities:

i. Initialize an empty doubly linked list.
ii. Insert a new node at the beginning of the list.
iii. Insert a new node at the end of the list.
iv. Insert a new node at a given position.
v. Delete a node with a given value from the list.
vi. Display the elements of the list.

You need to define suitable functions to perform these operations and demonstrate their usage in your program. Ensure that you handle edge cases, such as deleting a node that doesn't exist in the list or displaying an empty list.

Program

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

// Structure to represent a doubly linked list node
struct Node
{
    int data;
    struct Node* prev;
    struct Node* next;
};

// Structure to represent a doubly linked list
struct DoublyLinkedList
{
    struct Node* head;
};

// Function to initialize an empty doubly linked list
void initializeList(struct DoublyLinkedList* list)
{
    list->head = NULL;
}

// Function to insert a new node at the beginning of the list
void insertAtBeginning(struct DoublyLinkedList* list, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = list->head;

    if (list->head != NULL)
    {
        list->head->prev = newNode;
    }

    list->head = newNode;
}

// Function to insert a new node at the end of the list
void insertAtEnd(struct DoublyLinkedList* list, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;

    if (list->head == NULL)
    {
        list->head = newNode;
        return;
    }

    struct Node* current = list->head;
    while (current->next != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->prev = current;
}

// Function to insert a new node at a given position
void insertAtPosition(struct DoublyLinkedList* list, int data, int position)
{
    if (position < 0)
    {
        printf("Invalid position.\n");
        return;
    }

    if (position == 0)
    {
        insertAtBeginning(list, data);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;

    struct Node* current = list->head;
    int currentPosition = 0;

    while (current != NULL && currentPosition < position)
    {
        current = current->next;
        currentPosition++;
    }

    if (current == NULL)
    {
        printf("Position out of range.\n");
        free(newNode);
        return;
    }

    newNode->prev = current->prev;
    newNode->next = current;

    if (current->prev != NULL)
    {
        current->prev->next = newNode;
    }
    else
    {
        list->head = newNode;
    }

    current->prev = newNode;
}

// Function to delete a node with a given value from the list
void deleteNode(struct DoublyLinkedList* list, int value)
{
    struct Node* current = list->head;

    while (current != NULL)
    {
        if (current->data == value)
        {
            if (current->prev != NULL)
            {
                current->prev->next = current->next;
            }
            else
            {
                list->head = current->next;
            }

            if (current->next != NULL)
            {
                current->next->prev = current->prev;
            }

            free(current);
            return;
        }

        current = current->next;
    }

    printf("Value %d not found in the list.\n", value);
}

// Function to display the elements of the list
void displayList(struct DoublyLinkedList* list)
{
    struct Node* current = list->head;

    if (current == NULL)
    {
        printf("The list is empty.\n");
        return;
    }

    printf("Doubly Linked List: ");
    while (current != NULL)
    {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main()
{
    struct DoublyLinkedList myList;
    initializeList(&myList);

    insertAtBeginning(&myList, 10);
    insertAtEnd(&myList, 20);
    insertAtPosition(&myList, 15, 1);
    displayList(&myList);

    deleteNode(&myList, 20);

    displayList(&myList);

    return 0;
}
```

7.b. (**Optional Question)**
You are tasked with designing a program to create an online shopping cart. The program should use a doubly linked list to manage the user's shopping cart, where each item is represented by its name, price, quantity, and a pointer to the next and previous items.

Your program should provide the following functionalities:
i. Initialize an empty shopping cart
ii. Add items to the cart: Allow the user to add items to the shopping cart by entering the item's name, price, and quantity.
iii. Remove items from the cart: Allow the user to remove items from the cart by entering the item's name. If the item is not found, display an appropriate message.
iv. Display the cart: Provide an option to display the items in the shopping cart in both forward and backward directions, including their names, prices, quantities, and total cost per item.
v. Calculate the total cost: Calculate and display the total cost of all items in the shopping cart.

Program

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

// Structure to represent an item in the shopping cart
struct CartItem
{
   char name[100];
   float price;
   int quantity;
   struct CartItem* next;
   struct CartItem* prev;
};

// Structure to represent the shopping cart
struct ShoppingCart
{
   struct CartItem* head;
   struct CartItem* tail;
};

// Function to initialize an empty shopping cart
void initializeCart(struct ShoppingCart* cart)
{
   cart->head = NULL;
   cart->tail = NULL;
}

// Function to add items to the cart
void addItem(struct ShoppingCart* cart, const char* name, float price, int quantity)
{
   struct CartItem* newItem = (struct CartItem*)malloc(sizeof(struct CartItem));
```

```
    strncpy(newItem->name, name, sizeof(newItem->name));
    newItem->price = price;
    newItem->quantity = quantity;
    newItem->next = NULL;
    newItem->prev = cart->tail;

    if (cart->tail != NULL)
    {
        cart->tail->next = newItem;
    }

    cart->tail = newItem;

    if (cart->head == NULL)
    {
        cart->head = newItem;
    }
}

// Function to remove items from the cart by name
void removeItem(struct ShoppingCart* cart, const char* name)
{
    struct CartItem* current = cart->head;

    while (current != NULL)
    {
        if (strcmp(current->name, name) == 0)
        {
            if (current->prev != NULL)
            {
                current->prev->next = current->next;
            }
            else
            {
                cart->head = current->next;
            }

            if (current->next != NULL)
            {
                current->next->prev = current->prev;
            }
            else
            {
                cart->tail = current->prev;
            }

            free(current);
            return;
        }
        current = current->next;
    }

    printf("Item '%s' not found in the cart.\n", name);
}

// Function to display the cart in forward direction
void displayCartForward(struct ShoppingCart* cart)
{
    struct CartItem* current = cart->head;

    printf("Shopping Cart:\n");
    while (current != NULL)
    {
        printf("Name: %s, Price: $%.2f, Quantity: %d, Total Cost: $%.2f\n", current->name,
current->price, current->quantity, current->price * current->quantity);
        current = current->next;
    }
    printf("\n");
}

// Function to calculate and display the total cost of items in the cart
void displayTotalCost(struct ShoppingCart* cart)
{
    struct CartItem* current = cart->head;
    float totalCost = 0.0;

    while (current != NULL)
    {
        totalCost += (current->price * current->quantity);
        current = current->next;
    }

    printf("Total Cost of Items in the Cart: $%.2f\n", totalCost);
}

int main()
{
    struct ShoppingCart myCart;
    initializeCart(&myCart);

    // Add items to the cart
    addItem(&myCart, "Product A", 10.99, 2);
    addItem(&myCart, "Product B", 5.49, 3);
    addItem(&myCart, "Product C", 2.99, 1);

    // Display the cart
    displayCartForward(&myCart);

    // Remove an item from the cart
    removeItem(&myCart, "Product B");

    // Display the updated cart
    displayCartForward(&myCart);

    // Calculate and display the total cost
    displayTotalCost(&myCart);

    return 0;
}
```

Date:__ / __ / ____

Experiment 8

8. a. Write a C program to create a circular queue using a circular linked list data structure. Your program should support the following operations:

Enqueue (Insertion): Allow users to add elements to the circular queue.
Dequeue (Removal): Implement the ability to remove elements from the circular queue.

Ensure that your circular queue maintains its circular behavior, meaning that enqueuing elements beyond the queue's capacity should wrap around to the beginning. Additionally, handle cases when the queue is empty or full gracefully.

Program

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

// Define the structure for a queue node
struct QueueNode {
    int data;
    struct QueueNode* next;
};

// Define the circular queue structure
struct CircularQueue {
    struct QueueNode* front;
    struct QueueNode* rear;
};

// Function to create an empty circular queue
struct CircularQueue* createCircularQueue() {
    struct CircularQueue* queue = (struct CircularQueue*)malloc(sizeof(struct
CircularQueue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    queue->front = NULL;
    queue->rear = NULL;
    return queue;
}

// Function to check if the circular queue is empty
int isEmpty(struct CircularQueue* queue) {
    return (queue->front == NULL);
```

```
}

// Function to enqueue an element into the circular queue
void enqueue(struct CircularQueue* queue, int value) {
    struct QueueNode* newNode = (struct QueueNode*)malloc(sizeof(struct QueueNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (isEmpty(queue)) {
        // If the queue is empty, set both front and rear to the new node
        queue->front = newNode;
        queue->rear = newNode;
        newNode->next = newNode; // Circular link to itself
    } else {
        // Link the new node to the rear and update the rear
        newNode->next = queue->front; // Circular link
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Function to dequeue an element from the circular queue
int dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1; // Return a sentinel value for an empty queue
    }

    int removedValue;
    struct QueueNode* temp = queue->front;

    if (queue->front == queue->rear) {
        // If there is only one element in the queue
        removedValue = temp->data;
        queue->front = NULL;
        queue->rear = NULL;
        free(temp);
    } else {
        // Remove and update the front of the queue
        removedValue = temp->data;
        queue->front = temp->next;
        queue->rear->next = queue->front; // Circular link
        free(temp);
    }
    return removedValue;
}

// Function to display the elements in the circular queue
void displayQueue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    struct QueueNode* current = queue->front;
    printf("Circular Queue: ");
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != queue->front);
    printf(" (front)\n");
}

int main() {
    struct CircularQueue* queue = createCircularQueue();
    int choice, value;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter a value to enqueue: ");
                scanf("%d", &value);
                enqueue(queue, value);
                break;
            case 2:
                value = dequeue(queue);
                if (value != -1) {
                    printf("Dequeued value: %d\n", value);
                }
                break;
            case 3:
                displayQueue(queue);
                break;
            case 4:
                // Clean up and exit
                while (!isEmpty(queue)) {
                    dequeue(queue);
                }
                free(queue);
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

8. b. Develop a C program to create and manage a music playlist using a circular linked list. Your program should provide the following functionalities:

Operations
i. Add a Song
ii. Skip to Next Song
iii. Remove Currently Playing Song
iv. Display Currently Playing Song
v. Display Playlist

Add a Song: Allow users to add songs to the playlist. Each song has a title and artist name. New songs should be added to the end of the playlist.

Skip to Next Song: Implement an option for users to skip to the next song in the playlist. If the current song is the last one, the program should loop back to the first song.

Remove Currently Playing Song: Users should be able to remove the currently playing song from the playlist. After removal, the program should automatically start playing the next song.

Ensure that your program utilizes a circular linked list data structure to manage the playlist and maintains the circular behavior, allowing continuous music playback. Your program should display the playlist's current status and give users a menu to choose the above options.

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

// Define the structure for a song node
struct Song {
    char title[100];
    char artist[100];
    struct Song* next;
};

// Function to create a new song node
struct Song* createSong(const char* title, const char* artist) {
    struct Song* newSong = (struct Song*)malloc(sizeof(struct Song));
    if (newSong == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    strncpy(newSong->title, title, sizeof(newSong->title));
    strncpy(newSong->artist, artist, sizeof(newSong->artist));
    newSong->next = NULL;
    return newSong;
}
```

```
// Function to add a song to the end of the playlist
struct Song* addSong(struct Song* playlist, const char* title, const char* artist) {
    struct Song* newSong = createSong(title, artist);
    if (newSong == NULL) {
        return playlist;
    }

    if (playlist == NULL) {
        playlist = newSong;
        playlist->next = playlist; // Circular link to itself
    } else {
        struct Song* current = playlist;
        while (current->next != playlist) {
            current = current->next;
        }
        current->next = newSong;
        newSong->next = playlist;
    }

    return playlist;
}

// Function to skip to the next song
struct Song* skipToNext(struct Song* playlist) {
    if (playlist == NULL) {
        printf("Playlist is empty.\n");
        return playlist;
    }

    playlist = playlist->next; // Move to the next song
    return playlist;
}

// Function to remove the currently playing song
struct Song* removeCurrentSong(struct Song* playlist) {
    if (playlist == NULL) {
        printf("Playlist is empty.\n");
        return playlist;
    }

    if (playlist->next == playlist) {
        // Only one song in the playlist
        free(playlist);
        return NULL;
    }

    struct Song* current = playlist;
    while (current->next != playlist) {
        current = current->next;
    }

    struct Song* temp = playlist;
    current->next = playlist->next; // Remove the current song from the circular list
    playlist = playlist->next; // Move to the next song
    free(temp);

    return playlist;
}

// Function to display the currently playing song
void displayCurrentSong(struct Song* playlist) {
    if (playlist == NULL) {
        printf("Playlist is empty.\n");
    } else {
        printf("Currently Playing: %s by %s\n", playlist->title, playlist->artist);
    }
}

// Function to display the entire playlist
void displayPlaylist(struct Song* playlist) {
    if (playlist == NULL) {
        printf("Playlist is empty.\n");
        return;
    }

    struct Song* current = playlist;
    do {
        printf("%s by %s\n", current->title, current->artist);
        current = current->next;
    } while (current != playlist);
}

int main() {
    struct Song* playlist = NULL;
    int choice;
    char title[100], artist[100];

    while (1) {
        printf("\nMenu:\n");
        printf("1. Add a Song\n");
        printf("2. Skip to Next Song\n");
        printf("3. Remove Currently Playing Song\n");
        printf("4. Display Currently Playing Song\n");
        printf("5. Display Playlist\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the title of the song: ");
                scanf(" %[^\n]s", title);
                printf("Enter the artist of the song: ");
                scanf(" %[^\n]s", artist);
                playlist = addSong(playlist, title, artist);
                break;
            case 2:
                playlist = skipToNext(playlist);
                break;
            case 3:
                playlist = removeCurrentSong(playlist);
                break;
            case 4:
                displayCurrentSong(playlist);
                break;
            case 5:
                displayPlaylist(playlist);
                break;
            case 6:
                // Clean up and exit
                while (playlist != NULL) {
                    playlist = removeCurrentSong(playlist);
                }
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Date:__ / __ / ____

Experiment 9

9. a. Implement a program in C that demonstrates various operations on a binary search tree (BST). Your program should provide the following functionalities:

i. Initialize an empty BST: Initialize an empty binary search tree.
ii. Insert a node: Allow the user to insert nodes into the BST by entering integer values.
iii. Delete a node: Allow the user to delete a node from the BST by entering a value to be deleted. If the value is not found, display an appropriate message.
iv. Find Minimum and Maximum: Implement functions to find and display the minimum and maximum values in the BST.

Program

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

// Structure to represent a node in the BST
struct Node
{
   int data;
   struct Node* left;
   struct Node* right;
};

// Function to initialize an empty BST
struct Node* initializeBST()
{
   return NULL;
}

// Function to insert a node into the BST
struct Node* insertNode(struct Node* root, int value)
{
   if (root == NULL)
   {
      struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
      newNode->data = value;
      newNode->left = newNode->right = NULL;
      return newNode;
   }

   if (value < root->data)
   {
      root->left = insertNode(root->left, value);
```

```
    }
    else if (value > root->data)
    {
        root->right = insertNode(root->right, value);
    }

    return root;
}

// Function to find the minimum value in the BST
int findMinimum(struct Node* root)
{
    if (root == NULL)
    {
        printf("The BST is empty.\n");
        return -1;
    }

    while (root->left != NULL)
    {
        root = root->left;
    }

    return root->data;
}

// Function to find the maximum value in the BST
int findMaximum(struct Node* root)
{
    if (root == NULL)
    {
        printf("The BST is empty.\n");
        return -1;
    }

    while (root->right != NULL)
    {
        root = root->right;
    }

    return root->data;
}

// Function to delete a node from the BST
struct Node* deleteNode(struct Node* root, int value)
{
    if (root == NULL)
    {
        printf("Value %d not found in the BST.\n", value);
        return root;
    }

    if (value < root->data)
    {
        root->left = deleteNode(root->left, value);
    }
    else if (value > root->data)
    {
        root->right = deleteNode(root->right, value);
    }
    else
    {
        if (root->left == NULL)
        {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = root->right;
        while (temp->left != NULL)
        {
            temp = temp->left;
        }

        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

// Function to display the BST in inorder traversal
void displayInorder(struct Node* root)
{
    if (root == NULL)
    {
        return;
    }

    displayInorder(root->left);
    printf("%d ", root->data);
    displayInorder(root->right);
}
```

```
int main()
{
   struct Node* root = initializeBST();

   while (1)
   {
      printf("\nBinary Search Tree Menu:\n");
      printf("1. Insert a node\n");
      printf("2. Delete a node\n");
      printf("3. Find Minimum\n");
      printf("4. Find Maximum\n");
      printf("5. Display Inorder Traversal\n");
      printf("6. Exit\n");

      int choice, value;

      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice)
      {
         case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            root = insertNode(root, value);
            break;

         case 2:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            break;

         case 3:
            printf("Minimum value: %d\n", findMinimum(root));
            break;

         case 4:
            printf("Maximum value: %d\n", findMaximum(root));
            break;

         case 5:
            printf("Inorder Traversal: ");
            displayInorder(root);
            printf("\n");
            break;

         case 6:
            exit(0);
            default:
                printf("Invalid choice. Try again.\n");
                break;
        }
    }

    return 0;
}
```

Date:__ / __ / ____

Open Ended Questions

Instructions
1. The questions may be distributed at the start of the semester.
2. Students are encouraged to employ various data structure programming concepts to address the open-ended questions.
3. It is expected that students will solve and submit their solutions to the faculty members responsible for the course only after the conclusion of the second mid-semester exam.

Questions
1. A man in an automobile search for another man who is located at some point of a certain road. He starts at a given point and knows in advance the probability that the second man is at any given point of the road. Since the man being sought might be in either direction from the starting point, the searcher will, in general, must turn around many times before finding his target. How does he search to minimize the expected distance travelled? When can this minimum expectation be achieved?

2. The computing resources of a cloud are pooled and allocated according to customer demand. This has led to increased use of energy on the part of the service providers due to the need to maintain the computing infrastructure. What data structure will you use for allocating resources which addresses the issue of energy saving? Why? Design the solution.

3. Mini-Project on applying suitable data structure to a given real-world problem. (team of 4 can pair do the project).