Data Structures & Algorithms Lab

End Sem Lab Exam

PCC IT 391

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

B. TECH

in

INFORMATION TECHNOLOGY

Submitted by: - Anish Kumar

Roll No.: - 10000222005

Registration No.: 221000110031

<u>Course Faculty</u>

Department of Information Technology

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

Simhat, Haringhata, Nadia, Pin-741249

INDEX

Assignment - 1

1. **Please find the average marks of student (using static Array).**

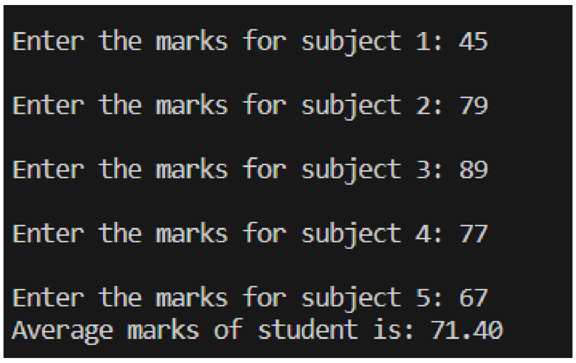
⇨ **Code:**

```c
#include <stdio.h>

#define SUBJECTS 5  // Define the number of subjects

int main() {
    int marks[SUBJECTS];

    // Input marks for each subject
    for (int i = 0; i < SUBJECTS; ) {
        int mark;
        do {
            printf("\nEnter the marks for subject %d: ", i + 1);
            if (scanf("%d", &mark) == 1) {
                if (mark >= 0 && mark <= 100) {
                    marks[i] = mark;
                    i++;
                } else {
                    printf("Invalid mark. Please enter a mark between 0 and 100.\n");
                }
            } else {
                printf("Invalid input. Please input a valid integer mark.\n");
                while (getchar() != '\n');
            }
        } while (mark < 0 || mark > 100);
    }

    // Calculating the sum of marks
    int sum = 0;
    for (int i = 0; i < SUBJECTS; i++) {
        sum += marks[i];
    }

    // Calculating the average
    double average = (double)sum / SUBJECTS;
    printf("Average marks of student is: %.2f\n\n", average);

    return 0;
}
```

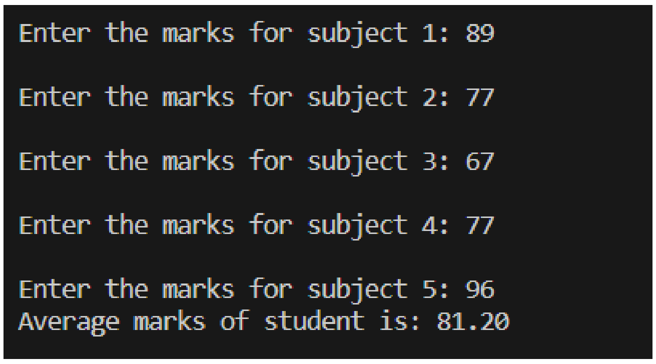
⇨ **Explanation:**

1. Static Array is an array whose size is determined at compile time.
2. marks[SUBJECTS] is a static array of size 5.
3. SUBJECTS is a global constant with value 5.

⇨ **Output:**



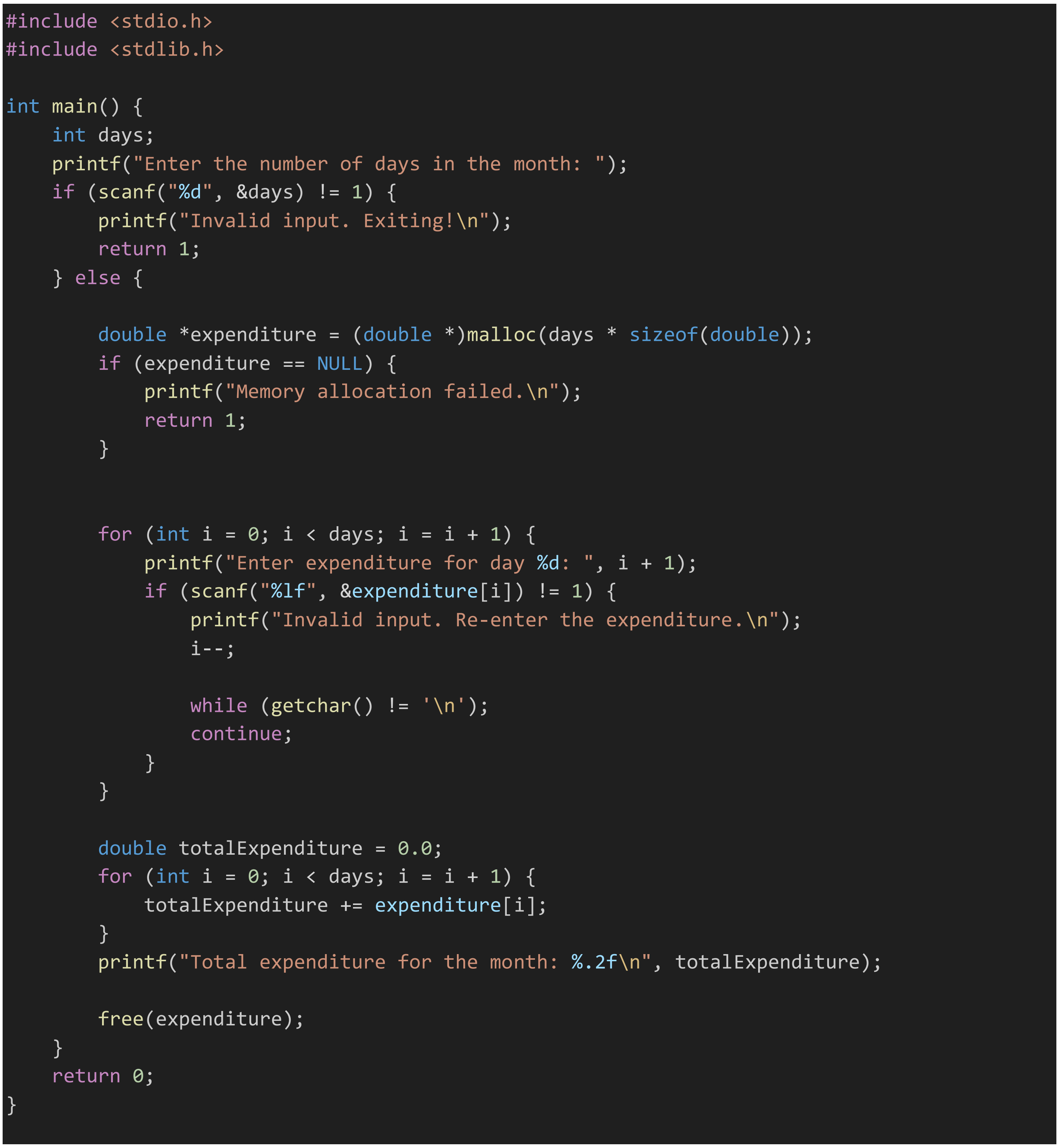
```
Enter the marks for subject 1: 45

Enter the marks for subject 2: 79

Enter the marks for subject 3: 89

Enter the marks for subject 4: 77

Enter the marks for subject 5: 67
Average marks of student is: 71.40
```



```
Enter the marks for subject 1: 89

Enter the marks for subject 2: 77

Enter the marks for subject 3: 67

Enter the marks for subject 4: 77

Enter the marks for subject 5: 96
Average marks of student is: 81.20
```

2. **Please find the total expenditure in a month (using dynamic array).**

⇨ **Code:**

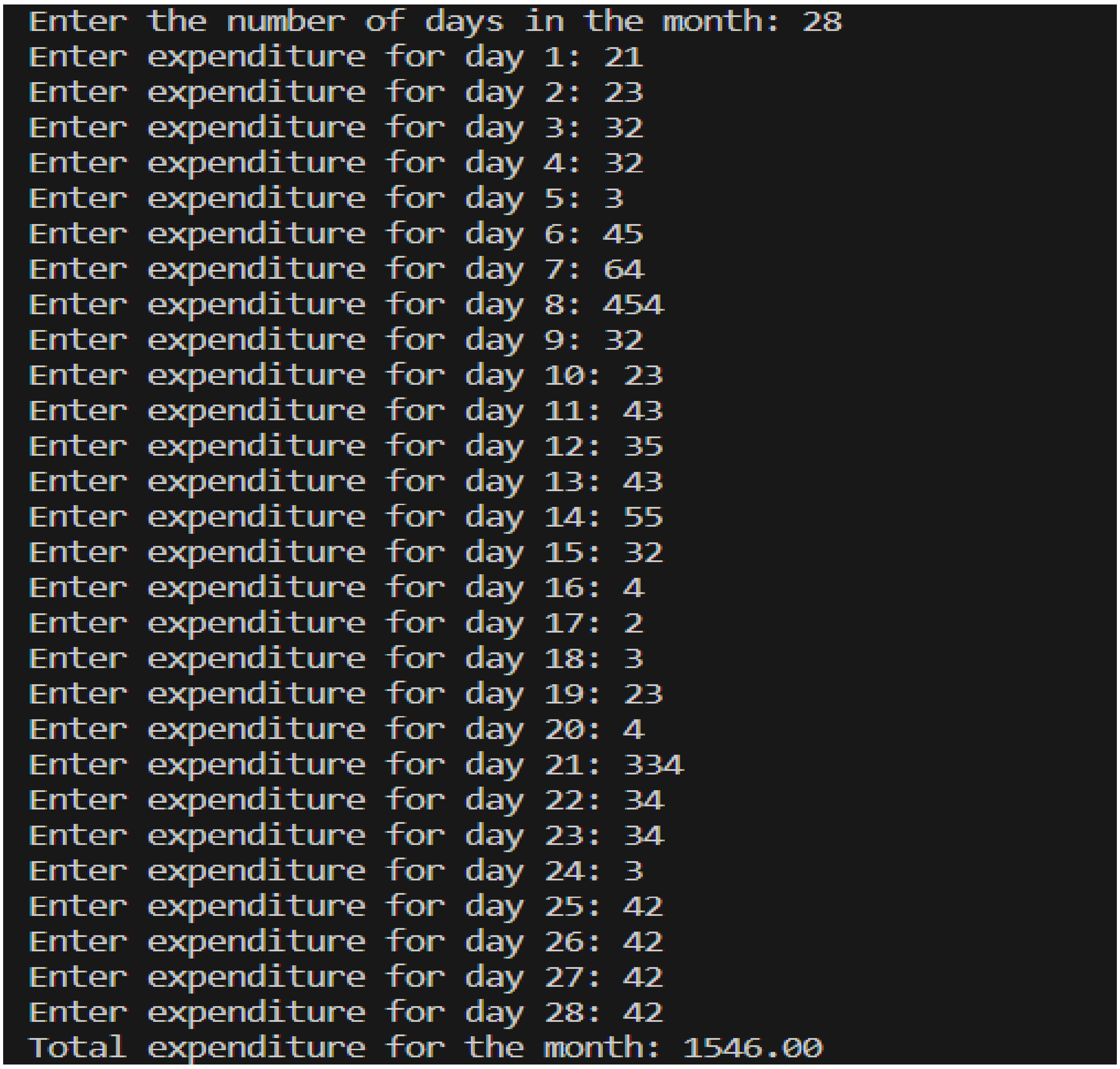


```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int days;
    printf("Enter the number of days in the month: ");
    if (scanf("%d", &days) != 1) {
        printf("Invalid input. Exiting!\n");
        return 1;
    } else {

        double *expenditure = (double *)malloc(days * sizeof(double));
        if (expenditure == NULL) {
            printf("Memory allocation failed.\n");
            return 1;
        }


        for (int i = 0; i < days; i = i + 1) {
            printf("Enter expenditure for day %d: ", i + 1);
            if (scanf("%lf", &expenditure[i]) != 1) {
                printf("Invalid input. Re-enter the expenditure.\n");
                i--;

                while (getchar() != '\n');
                continue;
            }
        }

        double totalExpenditure = 0.0;
        for (int i = 0; i < days; i = i + 1) {
            totalExpenditure += expenditure[i];
        }
        printf("Total expenditure for the month: %.2f\n", totalExpenditure);

        free(expenditure);
    }
    return 0;
}
```

⇨ **Explanation:**

1. Dynamic array is the array whose size is determined at runtime.
2. For that purpose, we can use dynamic memory allocation.
3. We should free the memory in order to maintain memory safety.

⇨ **Output:**



```
Enter the number of days in the month: 28
Enter expenditure for day 1: 21
Enter expenditure for day 2: 23
Enter expenditure for day 3: 32
Enter expenditure for day 4: 32
Enter expenditure for day 5: 3
Enter expenditure for day 6: 45
Enter expenditure for day 7: 64
Enter expenditure for day 8: 454
Enter expenditure for day 9: 32
Enter expenditure for day 10: 23
Enter expenditure for day 11: 43
Enter expenditure for day 12: 35
Enter expenditure for day 13: 43
Enter expenditure for day 14: 55
Enter expenditure for day 15: 32
Enter expenditure for day 16: 4
Enter expenditure for day 17: 2
Enter expenditure for day 18: 3
Enter expenditure for day 19: 23
Enter expenditure for day 20: 4
Enter expenditure for day 21: 334
Enter expenditure for day 22: 34
Enter expenditure for day 23: 34
Enter expenditure for day 24: 3
Enter expenditure for day 25: 42
Enter expenditure for day 26: 42
Enter expenditure for day 27: 42
Enter expenditure for day 28: 42
Total expenditure for the month: 1546.00
```

⇨ **Compilation and Execution:**

1. To compile use:
   gcc filename.c

2. To execute use:
   ./a.exe (on Windows), ./a.out (on Linux)

Assignment 2
Linked List

A linked list is a dynamic data structure that consists of nodes, where each node contains data and a reference (or link) to the next node in the sequence. Linked lists provide flexibility in memory management and efficient insertion and deletion of elements compared to arrays. Let's delve into the creation, traversal, insertion, and deletion operations, along with complexity analysis.

⇨ Node Structure:

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

⇨ Linked List Creation:

Creating a linked list involves allocating memory for each node and connecting them.

```
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

struct Node* createLinkedList(int values[], int length) {
    if (length == 0) {
        return NULL;
    }

    struct Node* head = createNode(values[0]);
    struct Node* current = head;

    for (int i = 1; i < length; i++) {
        current->next = createNode(values[i]);
        current = current->next;
    }

    return head;
}
```

⇨ Linked List Traversal:

Traversing the linked list involves visiting each node, starting from the head and continuing until the end.

```c
void traverseLinkedList(struct Node* head) {
    struct Node* current = head;

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

    printf("NULL\n");
}
```

⇨ Linked List Insertion:

1. Insertion at the Beginning:

```c
struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}
```

2. Insertion at a Specific Position:

```c
struct Node* insertAtPosition(struct Node* head, int value, int position) {
    struct Node* newNode = createNode(value);

    if (position == 1) {
        newNode->next = head;
        return newNode;
    }

    struct Node* current = head;
    for (int i = 1; i < position - 1 && current != NULL; i++) {
        current = current->next;
    }

    if (current == NULL) {
        printf("Invalid position for insertion.\n");
        return head;
    }

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

    return head;
}
```

3. Insertion at the End:

```c
struct Node* insertAtEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);

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

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

    current->next = newNode;

    return head;
}
```

⇨ Linked List Deletion:

1. Deletion from the Beginning:

```c
struct Node* deleteFromBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    struct Node* newHead = head->next;
    free(head);
    return newHead;
}
```

2. Deletion from a Specific Position:

```c
struct Node* deleteFromPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    if (position == 1) {
        struct Node* newHead = head->next;
        free(head);
        return newHead;
    }

    struct Node* current = head;
    struct Node* previous = NULL;

    for (int i = 1; i < position && current != NULL; i++) {
        previous = current;
        current = current->next;
    }

    if (current == NULL) {
        printf("Invalid position for deletion.\n");
        return head;
    }

    previous->next = current->next;
    free(current);

    return head;
}
```

3. Deletion from the End:

```
      struct Node* deleteFromEnd(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    if (head->next == NULL) {
        free(head);
        return NULL;
    }

    struct Node* current = head;
    struct Node* previous = NULL;

    while (current->next != NULL) {
        previous = current;
        current = current->next;
    }

    previous->next = NULL;
    free(current);

    return head;
}
```

⇨ **Linked List Complexity Analysis:**

- **Time Complexity:**

1. **Creation:** O(n) - Creating a linked list involves iterating through each element.
2. **Traversing:** O(n) - Visiting each element requires linear time.
3. **Insertion at Beginning/Position/End:** O(n) - In the worst case, inserting at a specific position may require traversing the entire list.

4. **Deletion from Beginning/Position/End:** O(n) - Deletion may involve traversing the list to find the target position.

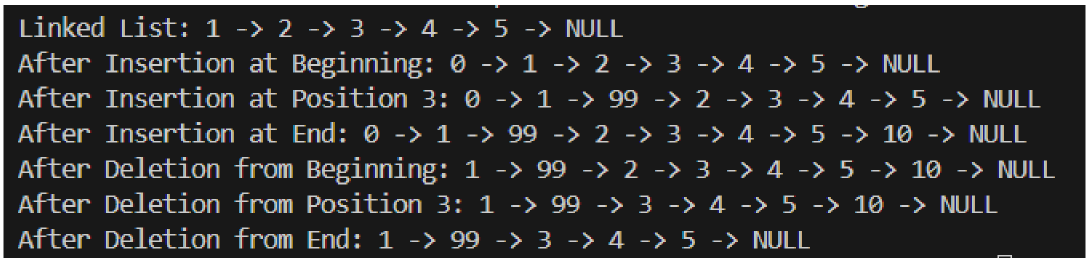
- **Space Complexity:**

1. **Node Creation:** O(1) - Creating each node requires constant space.
2. **Overall:** O(n) - The space required is proportional to the number of elements in the linked list.

⇨ **Output:**

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

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

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

struct Node* createLinkedList(int values[], int length) {
    if (length == 0) {
        return NULL;
    }

    struct Node* head = createNode(values[0]);
    struct Node* current = head;

    for (int i = 1; i < length; i++) {
        current->next = createNode(values[i]);
        current = current->next;
    }

    return head;
}

void traverseLinkedList(struct Node* head) {
    struct Node* current = head;

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

    printf("NULL\n");
}

struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}

struct Node* insertAtPosition(struct Node* head, int value, int position) {
    struct Node* newNode = createNode(value);

    if (position == 1) {
        newNode->next = head;
        return newNode;
    }

    struct Node* current = head;
    for (int i = 1; i < position - 1 && current != NULL; i++) {
        current = current->next;
    }

    if (current == NULL) {
        printf("Invalid position for insertion.\n");
        return head;
    }

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

    return head;
}

struct Node* insertAtEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);

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

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

    current->next = newNode;

    return head;
}

struct Node* deleteFromBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    struct Node* newHead = head->next;
    free(head);
    return newHead;
}

struct Node* deleteFromPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    if (position == 1) {
        struct Node* newHead = head->next;
        free(head);
        return newHead;
    }

    struct Node* current = head;
    struct Node* previous = NULL;

    for (int i = 1; i < position && current != NULL; i++) {
        previous = current;
        current = current->next;
    }

    if (current == NULL) {
        printf("Invalid position for deletion.\n");
        return head;
    }

    previous->next = current->next;
    free(current);

    return head;
}

struct Node* deleteFromEnd(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return NULL;
    }

    if (head->next == NULL) {
        free(head);
        return NULL;
    }

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

    previous->next = NULL;
    free(current);

    return head;
}

int main() {
    int values[] = {1, 2, 3, 4, 5};
    int length = sizeof(values) / sizeof(values[0]);

    struct Node* head = createLinkedList(values, length);

    printf("Linked List: ");
    traverseLinkedList(head);

    head = insertAtBeginning(head, 0);
    printf("After Insertion at Beginning: ");
    traverseLinkedList(head);

    head = insertAtPosition(head, 99, 3);
    printf("After Insertion at Position 3: ");
    traverseLinkedList(head);

    head = insertAtEnd(head, 10);
    printf("After Insertion at End: ");
    traverseLinkedList(head);

    head = deleteFromBeginning(head);
    printf("After Deletion from Beginning: ");
    traverseLinkedList(head);

    head = deleteFromPosition(head, 3);
    printf("After Deletion from Position 3: ");
    traverseLinkedList(head);

    head = deleteFromEnd(head);
    printf("After Deletion from End: ");
    traverseLinkedList(head);

    return 0;
}
```

⇨ **Output:**



```
Linked List: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
After Insertion at Beginning: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL
After Insertion at Position 3: 0 -> 1 -> 99 -> 2 -> 3 -> 4 -> 5 -> NULL
After Insertion at End: 0 -> 1 -> 99 -> 2 -> 3 -> 4 -> 5 -> 10 -> NULL
After Deletion from Beginning: 1 -> 99 -> 2 -> 3 -> 4 -> 5 -> 10 -> NULL
After Deletion from Position 3: 1 -> 99 -> 3 -> 4 -> 5 -> 10 -> NULL
After Deletion from End: 1 -> 99 -> 3 -> 4 -> 5 -> NULL
```

Assignment - 3

1. **Write a Program to store the marks obtained in all courses in a particular semester and then calculate the average marks obtained by the student.**

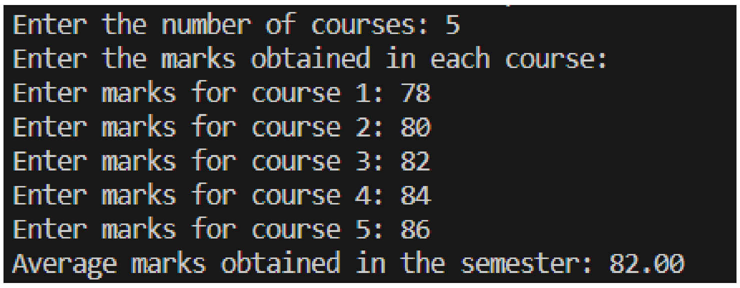
⇨ **Code:**

```c
#include <stdio.h>

#define MAX_COURSES 10

int main() {
    int marks[MAX_COURSES];
    int numCourses, i;
    float totalMarks = 0, averageMarks;


    printf("Enter the number of courses: ");
    scanf("%d", &numCourses);


    if (numCourses <= 0 || numCourses > MAX_COURSES) {
        printf("Invalid number of courses. Please enter a number between 1 and %d.\n",
MAX_COURSES);
        return 1;
    }


    printf("Enter the marks obtained in each course:\n");
    for (i = 0; i < numCourses; i++) {
        printf("Enter marks for course %d: ", i + 1);
        scanf("%d", &marks[i]);


        if (marks[i] < 0) {
            printf("Invalid marks. Marks cannot be negative.\n");
            return 1;
        }


        totalMarks += marks[i];
    }


    averageMarks = totalMarks / numCourses;


    printf("Average marks obtained in the semester: %.2f\n", averageMarks);

    return 0;
}
```

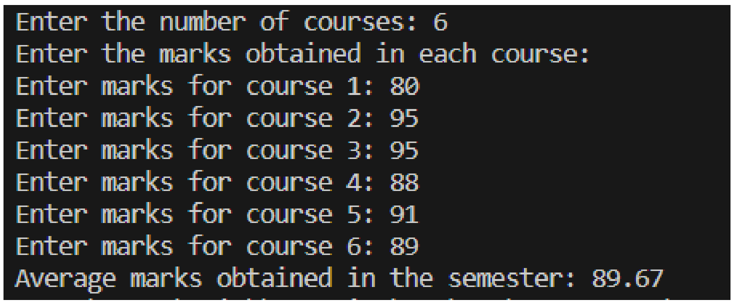
⇨ **Explanation:**

1. **Input Number of Courses:** The program prompts the user to enter the number of courses the student has taken in a particular semester. This input is stored in the variable numCourses.
2. **Input Marks for Each Course:** For each course, the program prompts the user to input the marks obtained. It stores these marks in an array called marks[]. Before storing each mark, the program checks if it is negative, and if so, it displays an error message and terminates.
3. **Calculate Average Marks:** After obtaining all the marks, the program calculates the total marks obtained by summing up all the marks in the marks[] array. Then, it calculates the average marks by dividing the total marks by the number of courses (numCourses). The result is stored in the variable averageMarks.
4. **Display Average Marks:** Finally, the program displays the average marks obtained by the student in the semester, rounded to two decimal places, using printf.

⇨ **Output:**



```
Enter the number of courses: 5
Enter the marks obtained in each course:
Enter marks for course 1: 78
Enter marks for course 2: 80
Enter marks for course 3: 82
Enter marks for course 4: 84
Enter marks for course 5: 86
Average marks obtained in the semester: 82.00
```



```
Enter the number of courses: 6
Enter the marks obtained in each course:
Enter marks for course 1: 80
Enter marks for course 2: 95
Enter marks for course 3: 95
Enter marks for course 4: 88
Enter marks for course 5: 91
Enter marks for course 6: 89
Average marks obtained in the semester: 89.67
```

2. **Write a Program to store the marks obtained in all courses along with credit of the course in a particular semester [example credit of DSA is 4. System and signal is 3 (as per syllabus)] and then calculate the SGPA marks obtained by the student.**

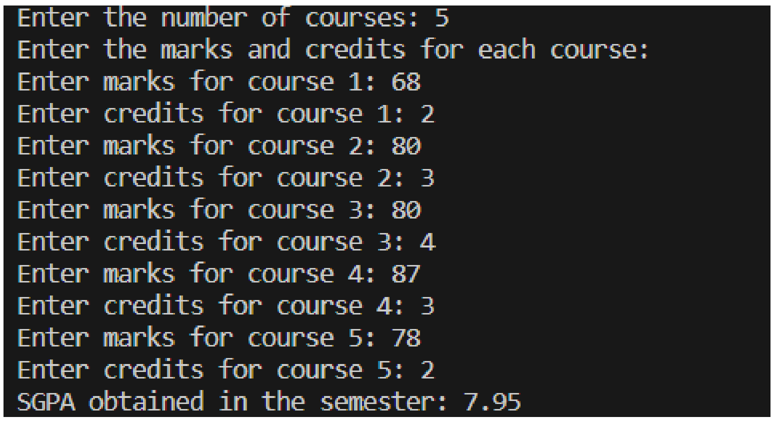
```c
#include <stdio.h>

#define MAX_COURSES 10

int main() {
    int marks[MAX_COURSES];
    int credits[MAX_COURSES];
    int numCourses, i;
    float totalMarks = 0, totalCredits = 0, sgpa;
    printf("Enter the number of courses: ");
    scanf("%d", &numCourses);


    if (numCourses <= 0 || numCourses > MAX_COURSES) {
        printf("Invalid number of courses. Please enter a number between 1 and %d.\n", 
MAX_COURSES);
        return 1;
    }


    printf("Enter the marks and credits for each course:\n");
    for (i = 0; i < numCourses; i++) {
        printf("Enter marks for course %d: ", i + 1);
        scanf("%d", &marks[i]);

        printf("Enter credits for course %d: ", i + 1);
        scanf("%d", &credits[i]);


        if (marks[i] < 0 || credits[i] <= 0) {
            printf("Invalid input. Marks cannot be negative and credits must be 
positive.\n");
            return 1;
        }


        totalMarks += marks[i] * credits[i];

        totalCredits += credits[i];
    }


    sgpa = totalMarks / (totalCredits*10);


    printf("SGPA obtained in the semester: %.2f\n", sgpa);

    return 0;
}
```

⇨ **Explanation:**

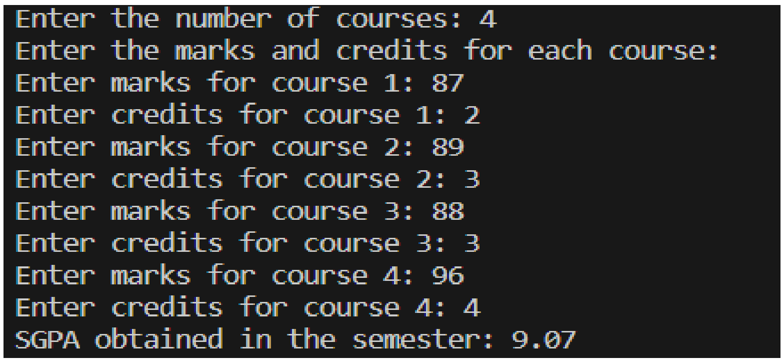
1. **Input**: It takes input for the number of courses, marks obtained, and credits for each course.
2. **Validation:** Validates the input to ensure it falls within acceptable ranges (positive marks, positive credits, and a reasonable number of courses).
3. **Calculation**: Calculates the total weighted marks by summing up marks multiplied by corresponding credits. Also calculates the total credits.

4. **SGPA Calculation:** Computes SGPA using the formula: SGPA = (Total weighted marks) / (Total credits * 10).
5. **Output:** Displays the calculated SGPA rounded to two decimal places.

⇨ **Output:**



```
Enter the number of courses: 5
Enter the marks and credits for each course:
Enter marks for course 1: 68
Enter credits for course 1: 2
Enter marks for course 2: 80
Enter credits for course 2: 3
Enter marks for course 3: 80
Enter credits for course 3: 4
Enter marks for course 4: 87
Enter credits for course 4: 3
Enter marks for course 5: 78
Enter credits for course 5: 2
SGPA obtained in the semester: 7.95
```



```
Enter the number of courses: 4
Enter the marks and credits for each course:
Enter marks for course 1: 87
Enter credits for course 1: 2
Enter marks for course 2: 89
Enter credits for course 2: 3
Enter marks for course 3: 88
Enter credits for course 3: 3
Enter marks for course 4: 96
Enter credits for course 4: 4
SGPA obtained in the semester: 9.07
```

3. **Write a program to add two numbers using the concept of pointer and functions.**

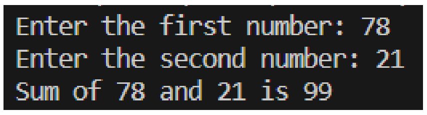
```
#include <stdio.h>


int addNumbers(int *a, int *b) {
    return *a + *b;
}

int main() {
    int num1, num2, sum;


    printf("Enter the first number: ");
    scanf("%d", &num1);


    printf("Enter the second number: ");
    scanf("%d", &num2);
```

```c
    sum = addNumbers(&num1, &num2);


    printf("Sum of %d and %d is %d\n", num1, num2, sum);

    return 0;
}
```

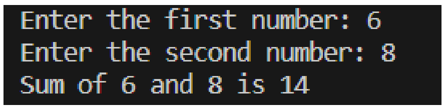
⇨ **Explanation:**

1. **Input**: The program prompts the user to enter two numbers.
2. **Function:** It defines a function addNumbers that takes two integer pointers as parameters and returns the sum of the values they point to.
3. **Calculation**: Inside the main function, it calls the addNumbers function, passing the addresses of the input numbers, and stores the result in the variable sum.
4. **Output**: Finally, it displays the sum of the two numbers along with the numbers themselves.

⇨ **Output:**



```
Enter the first number: 78
Enter the second number: 21
Sum of 78 and 21 is 99
```



```
Enter the first number: 6
Enter the second number: 8
Sum of 6 and 8 is 14
```

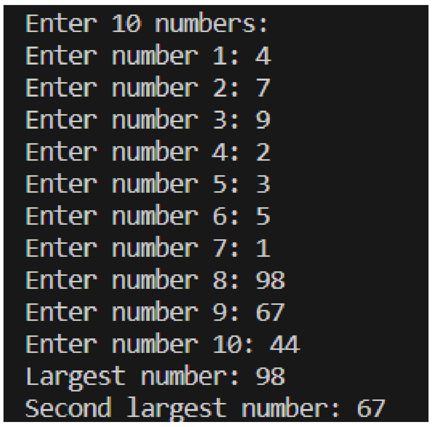
4. **Write a program to find the largest and second largest number from 10 given numbers.**

```c
#include <stdio.h>

int main() {
    int numbers[10];
    int i, largest, secondLargest;


    printf("Enter 10 numbers:\n");
    for (i = 0; i < 10; i++) {
        printf("Enter number %d: ", i + 1);
        scanf("%d", &numbers[i]);
    }


    if (numbers[0] > numbers[1]) {
        largest = numbers[0];
        secondLargest = numbers[1];
    } else {
        largest = numbers[1];
        secondLargest = numbers[0];
    }
    for (i = 2; i < 10; i++) {
        if (numbers[i] > largest) {
            secondLargest = largest;
            largest = numbers[i];
        } else if (numbers[i] > secondLargest && numbers[i] != largest) {
            secondLargest = numbers[i];
        }
    }


    printf("Largest number: %d\n", largest);
    printf("Second largest number: %d\n", secondLargest);

    return 0;
}
```

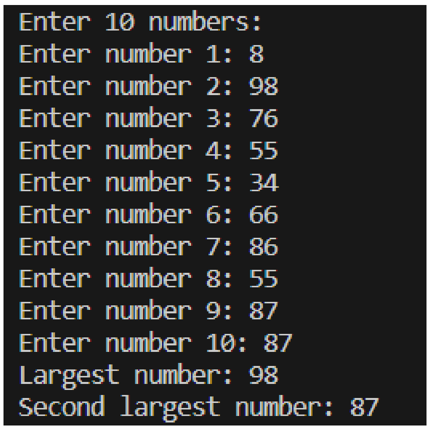
⇨ **Explanation:**

1. **Input**: The program prompts the user to enter 10 numbers and stores them in an array named numbers[].
2. **Initialization:** It initializes variables largest and secondLargest to store the largest and second largest numbers.
3. **Finding Largest and Second Largest:** It iterates through the array and compares each element with the current largest and second largest numbers. It updates largest and secondLargest accordingly.
4. **Output**: Finally, it displays the largest and second largest numbers found in the array.

⇨ **Output:**



```
Enter 10 numbers:
Enter number 1: 4
Enter number 2: 7
Enter number 3: 9
Enter number 4: 2
Enter number 5: 3
Enter number 6: 5
Enter number 7: 1
Enter number 8: 98
Enter number 9: 67
Enter number 10: 44
Largest number: 98
Second largest number: 67
```



```
Enter 10 numbers:
Enter number 1: 8
Enter number 2: 98
Enter number 3: 76
Enter number 4: 55
Enter number 5: 34
Enter number 6: 66
Enter number 7: 86
Enter number 8: 55
Enter number 9: 87
Enter number 10: 87
Largest number: 98
Second largest number: 87
```

5. **Write a program to transpose the given matrix.**

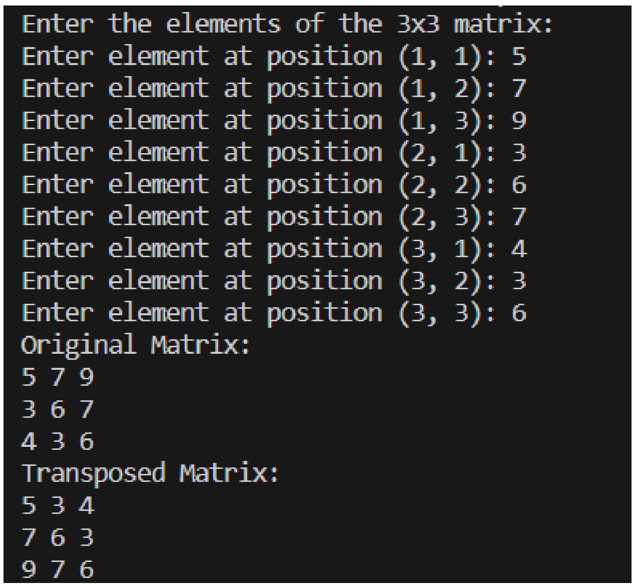
```
#include <stdio.h>

#define ROWS 3
#define COLS 3
```

```c
void transposeMatrix(int matrix[ROWS][COLS]) {
    int i, j;
    int temp;


    for (i = 0; i < ROWS; i++) {
        for (j = i + 1; j < COLS; j++) {

            temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}


void displayMatrix(int matrix[ROWS][COLS]) {
    int i, j;

    printf("Transposed Matrix:\n");
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int matrix[ROWS][COLS];
    int i, j;


    printf("Enter the elements of the %dx%d matrix:\n", ROWS, COLS);
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("Enter element at position (%d, %d): ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }


    printf("Original Matrix:\n");
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }


    transposeMatrix(matrix);

    displayMatrix(matrix);

    return 0;
}
```

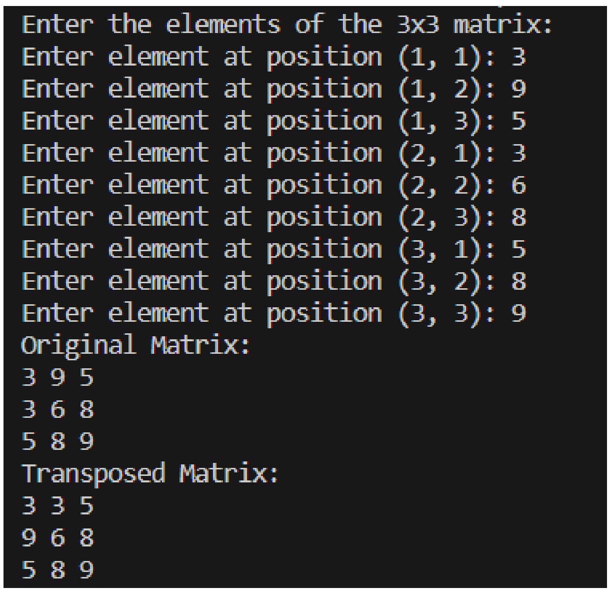
➪ **Explanation:**

1. **Input:** The program prompts the user to enter elements for a 3x3 matrix and stores them in the array matrix[][].
2. **Transpose Matrix Function:** It defines a function transposeMatrix to transpose the given matrix. It iterates through the upper triangular part of the matrix and swaps the elements across the diagonal.
3. **Display Matrix Function:** It defines a function displayMatrix to print the elements of the matrix.
4. **Main Function:** It prompts the user to input elements for the matrix and displays the original matrix. Then, it calls the transposeMatrix function to transpose the matrix. Finally, it displays the transposed matrix using the displayMatrix function.

➪ **Output:**



```
Enter the elements of the 3x3 matrix:
Enter element at position (1, 1): 5
Enter element at position (1, 2): 7
Enter element at position (1, 3): 9
Enter element at position (2, 1): 3
Enter element at position (2, 2): 6
Enter element at position (2, 3): 7
Enter element at position (3, 1): 4
Enter element at position (3, 2): 3
Enter element at position (3, 3): 6
Original Matrix:
5 7 9
3 6 7
4 3 6
Transposed Matrix:
5 3 4
7 6 3
9 7 6
```



```
Enter the elements of the 3x3 matrix:
Enter element at position (1, 1): 3
Enter element at position (1, 2): 9
Enter element at position (1, 3): 5
Enter element at position (2, 1): 3
Enter element at position (2, 2): 6
Enter element at position (2, 3): 8
Enter element at position (3, 1): 5
Enter element at position (3, 2): 8
Enter element at position (3, 3): 9
Original Matrix:
3 9 5
3 6 8
5 8 9
Transposed Matrix:
3 3 5
9 6 8
5 8 9
```

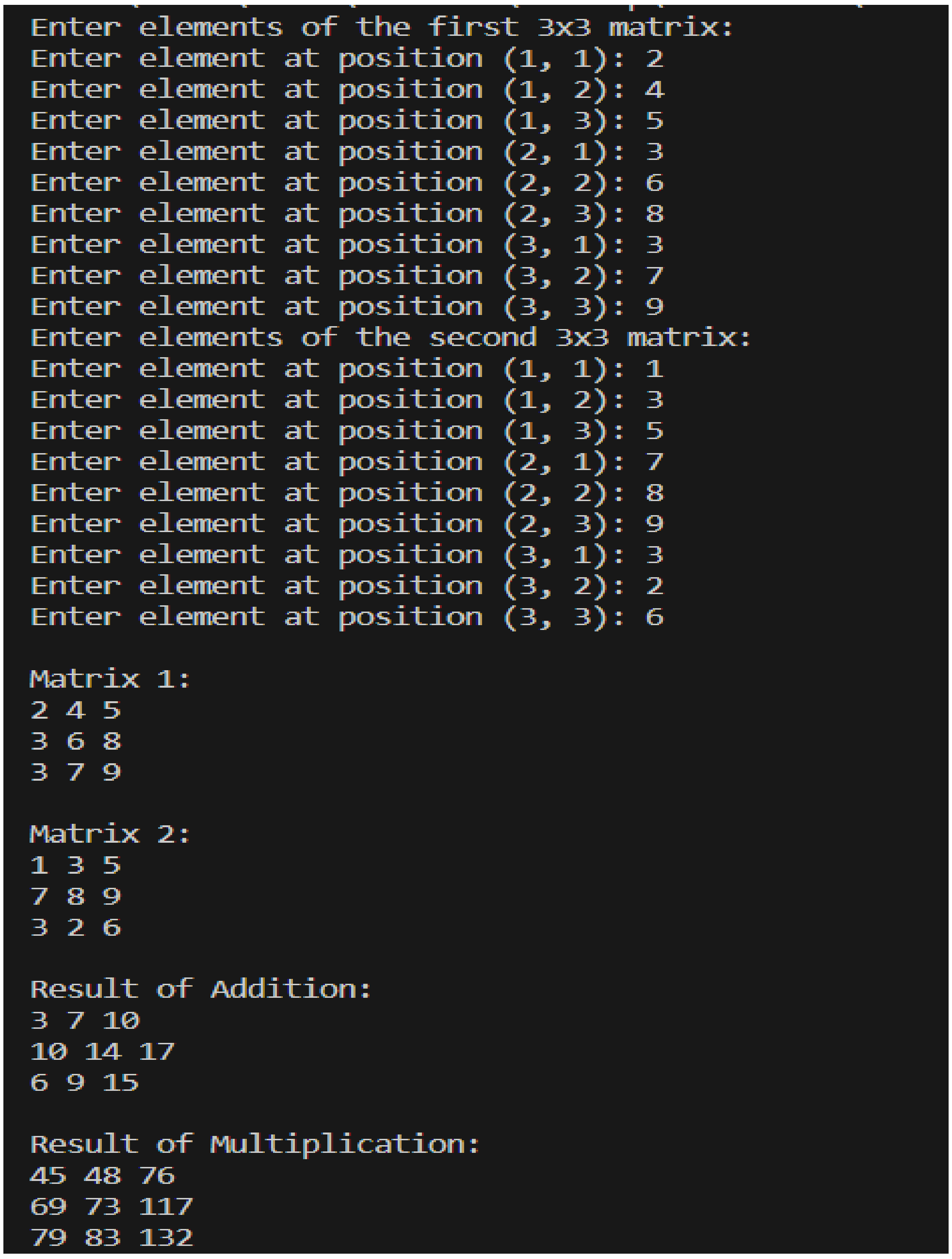
6. **Write A Program to Perform Addition, Multiplication operations on Matrix.**

```c
#include <stdio.h>

#define ROWS 3
#define COLS 3


void addMatrices(int mat1[ROWS][COLS], int mat2[ROWS][COLS], int result[ROWS][COLS]) {
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            result[i][j] = mat1[i][j] + mat2[i][j];
        }
    }
}


void multiplyMatrices(int mat1[ROWS][COLS], int mat2[ROWS][COLS], int result[ROWS][COLS])
{
    int i, j, k;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            result[i][j] = 0;
            for (k = 0; k < COLS; k++) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
}

// Function to display a matrix
void displayMatrix(int matrix[ROWS][COLS]) {
    int i, j;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int mat1[ROWS][COLS], mat2[ROWS][COLS], resultAdd[ROWS][COLS], resultMul[ROWS][COLS];
    int i, j;


    printf("Enter elements of the first %dx%d matrix:\n", ROWS, COLS);
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("Enter element at position (%d, %d): ", i + 1, j + 1);
            scanf("%d", &mat1[i][j]);
        }
    }


    printf("Enter elements of the second %dx%d matrix:\n", ROWS, COLS);
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("Enter element at position (%d, %d): ", i + 1, j + 1);
            scanf("%d", &mat2[i][j]);
        }
    }


    addMatrices(mat1, mat2, resultAdd);


    multiplyMatrices(mat1, mat2, resultMul);


    printf("\nMatrix 1:\n");
    displayMatrix(mat1);
    printf("\n");

    printf("Matrix 2:\n");
    displayMatrix(mat2);
    printf("\n");


    printf("Result of Addition:\n");
    displayMatrix(resultAdd);
    printf("\n");


    printf("Result of Multiplication:\n");
    displayMatrix(resultMul);

    return 0;
}
```

⇨ **Explanation:**

1. **Input:** The program prompts the user to input elements for two 3x3 matrices mat1 and mat2.
2. **Addition Function:** It defines a function addMatrices to perform addition operation on two matrices. It adds corresponding elements of mat1 and mat2 and stores the result in the resultAdd matrix.
3. **Multiplication Function:** It defines a function multiplyMatrices to perform multiplication operation on two matrices. It multiplies mat1 and mat2 using the standard matrix multiplication algorithm and stores the result in the resultMul matrix.
4. **Display Function:** It defines a function displayMatrix to print the elements of a matrix.
5. **Main Function:** It prompts the user to input elements for both matrices, then it performs addition and multiplication operations using the defined functions. Finally, it displays the original matrices along with the results of the addition and multiplication operations.

⇨ **Output:**



```
Enter elements of the first 3x3 matrix:
Enter element at position (1, 1): 2
Enter element at position (1, 2): 4
Enter element at position (1, 3): 5
Enter element at position (2, 1): 3
Enter element at position (2, 2): 6
Enter element at position (2, 3): 8
Enter element at position (3, 1): 3
Enter element at position (3, 2): 7
Enter element at position (3, 3): 9
Enter elements of the second 3x3 matrix:
Enter element at position (1, 1): 1
Enter element at position (1, 2): 3
Enter element at position (1, 3): 5
Enter element at position (2, 1): 7
Enter element at position (2, 2): 8
Enter element at position (2, 3): 9
Enter element at position (3, 1): 3
Enter element at position (3, 2): 2
Enter element at position (3, 3): 6

Matrix 1:
2 4 5
3 6 8
3 7 9

Matrix 2:
1 3 5
7 8 9
3 2 6

Result of Addition:
3 7 10
10 14 17
6 9 15

Result of Multiplication:
45 48 76
69 73 117
79 83 132
```

⇨ **Compile and Run:**

1. Use **gcc filename.c** to compile
2. Use **./a.exe** or **./a.out** to run

Assignment 4

Stack Implementation Using Linked List

⇨ **Introduction:**

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, which means that the last element added to the stack is the first one to be removed. Think of it like a stack of books, where you can only add or remove books from the top.

- In this program, I have used a linked list to create and manipulate a stack. A linked list is a data structure consisting of a collection of nodes, where each node contains data and a reference (or pointer) to the next node. In our case, each node represents an element in the stack.

⇨ **Here's what this program is capable of:**

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove and display the top element from the stack.
- **Display:** Show the elements currently in the stack.
- **Exit:** Terminate the program.

⇨ **Algorithm:**

- **Initialization:**

1. Initialize an empty linked list (top pointer set to NULL).
2. Initialize a variable stackSize to keep track of the number of elements in the stack.
3. Push Operation:

4. Create a new node and allocate memory for it.
5. Set the data of the new node to the value being pushed.
6. Set the next pointer of the new node to the current top of the stack.
7. Update the top pointer to the new node.
8. Increment stackSize.
9. Done.

- **Pop Operation:**

1. Check if the stack is empty (if top is NULL).
2. If empty, return an error or a sentinel value.
3. Otherwise, store the data of the current top node.

4. Update top to point to the next node.
5. Free the memory of the node being removed.
6. Decrement stackSize.
7. Return the stored data.
8. Done.

- **Display Operation:**

1. Traverse the linked list starting from top.
2. Print the data of each node until the end of the list.
3. Done.

- **Termination:**

1. Free any remaining memory if needed.
2. Done.

⇨ **Code:**

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

struct StackNode {
    int data;
    struct StackNode* next;
};

struct StackNode* top = NULL;
int stackSize = 0;

void push(int value) {
    struct StackNode* newNode = (struct StackNode*)malloc(sizeof(struct StackNode));
    if (newNode == NULL) {
        printf("Error: Memory allocation failed.\n");
        return;
    }

    newNode->data = value;
    newNode->next = top;
    top = newNode;
    stackSize++;
    printf("Element pushed onto the stack.\n");
}

int pop() {
    if (top == NULL) {
        printf("Error: Stack underflow.\n");
        return -1;
    }
    int poppedValue = top->data;
    struct StackNode* temp = top;
    top = top->next;
    free(temp);
    stackSize--;
    return poppedValue;
}

void displayStack() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }

    struct StackNode* current = top;
    printf("Stack elements: ");
    while (current != NULL) {
        printf("%d --> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, value;
    printf("\nStack Implementation using Linked List\n");
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");

        if (scanf("%d", &choice) != 1) {
            printf("Error: Invalid input. Please enter a valid integer choice.\n");
            while(getchar() != '\n');
            continue;
        }

        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
                if (scanf("%d", &value) != 1) {
                    printf("Error: Invalid input. Please enter a valid integer value.\n");
                    while(getchar() != '\n');
                } else {
                    push(value);
                }
                break;
            case 2:
                printf("Popped element is: %d\n", pop());
                break;
            case 3:
                displayStack();
                break;
            case 4:
                exit(0);
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}
```

⇨ **Complexity:**

- **Push Operation:**

1. Time Complexity: O(1)
Constant time since it involves creating a new node and updating pointers.

2. Space Complexity: O(1)
Constant space for the new node.

- **Pop Operation:**

1. Time Complexity: O(1)
Constant time since it involves updating pointers and freeing memory.

2. Space Complexity: O(1)
Constant space for the removed node.

- **Display Operation:**

1. Time Complexity: O(n)
Linear time, where n is the number of elements in the stack.

2. Space Complexity: O(1)
Constant space.

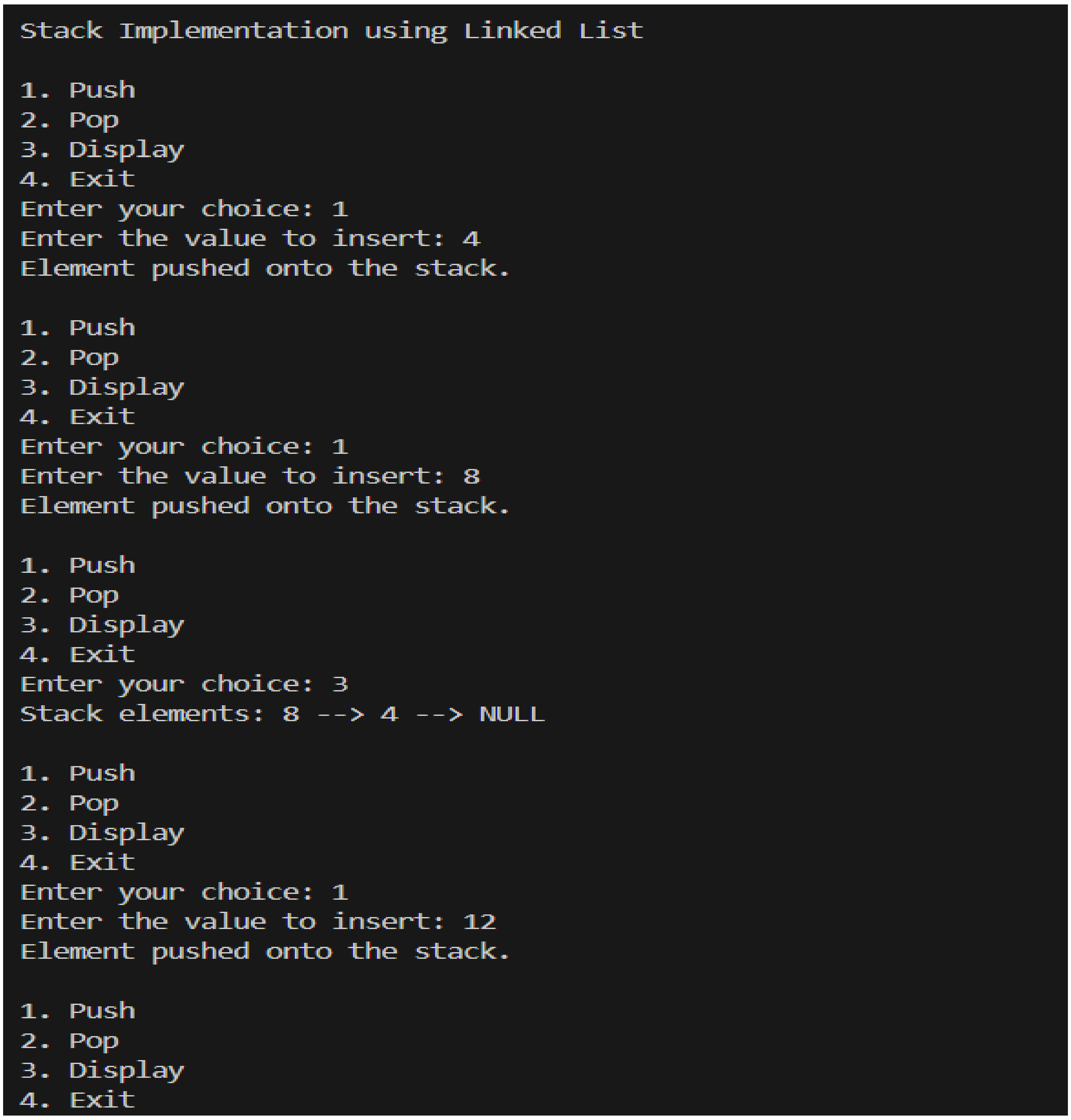
- **Overall Space Complexity: O(n)**

The space required grows linearly with the number of elements in the stack.

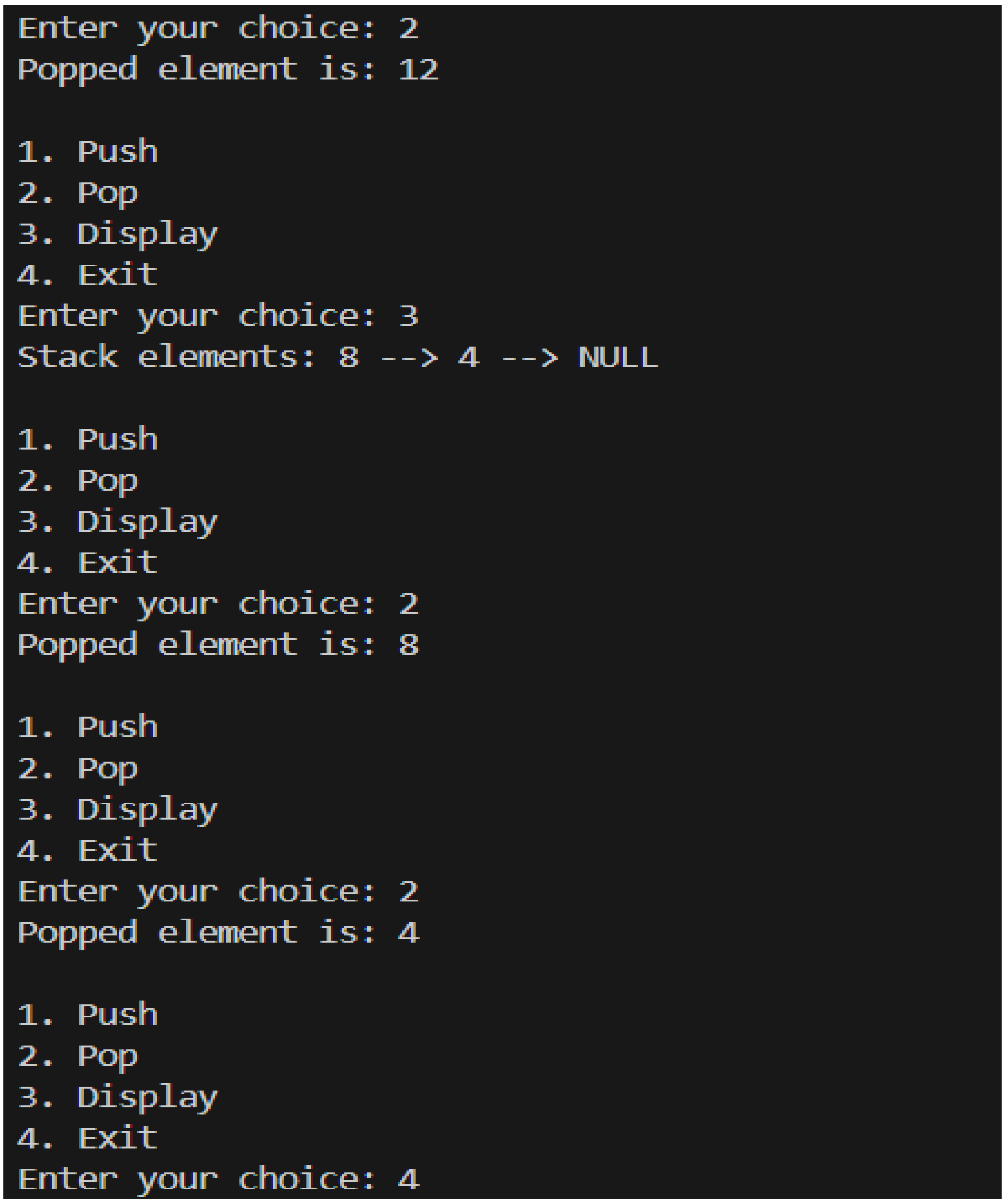
⇨ **To Compile and Run:**

- Use **gcc filename.c** to compile.
- Use **./a.exe** or **./a.out** to run based on the OS.

⇨ **Output:**



```
Stack Implementation using Linked List

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 4
Element pushed onto the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 8
Element pushed onto the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 8 --> 4 --> NULL

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 12
Element pushed onto the stack.

1. Push
2. Pop
3. Display
4. Exit
```


```
Enter your choice: 2
Popped element is: 12

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 8 --> 4 --> NULL

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped element is: 8

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped element is: 4

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

Assignment 5

Infix to Postfix Conversion Using Stack:

Infix Notation: Infix notation is the standard way of writing mathematical expressions. Operators are written between their operands.

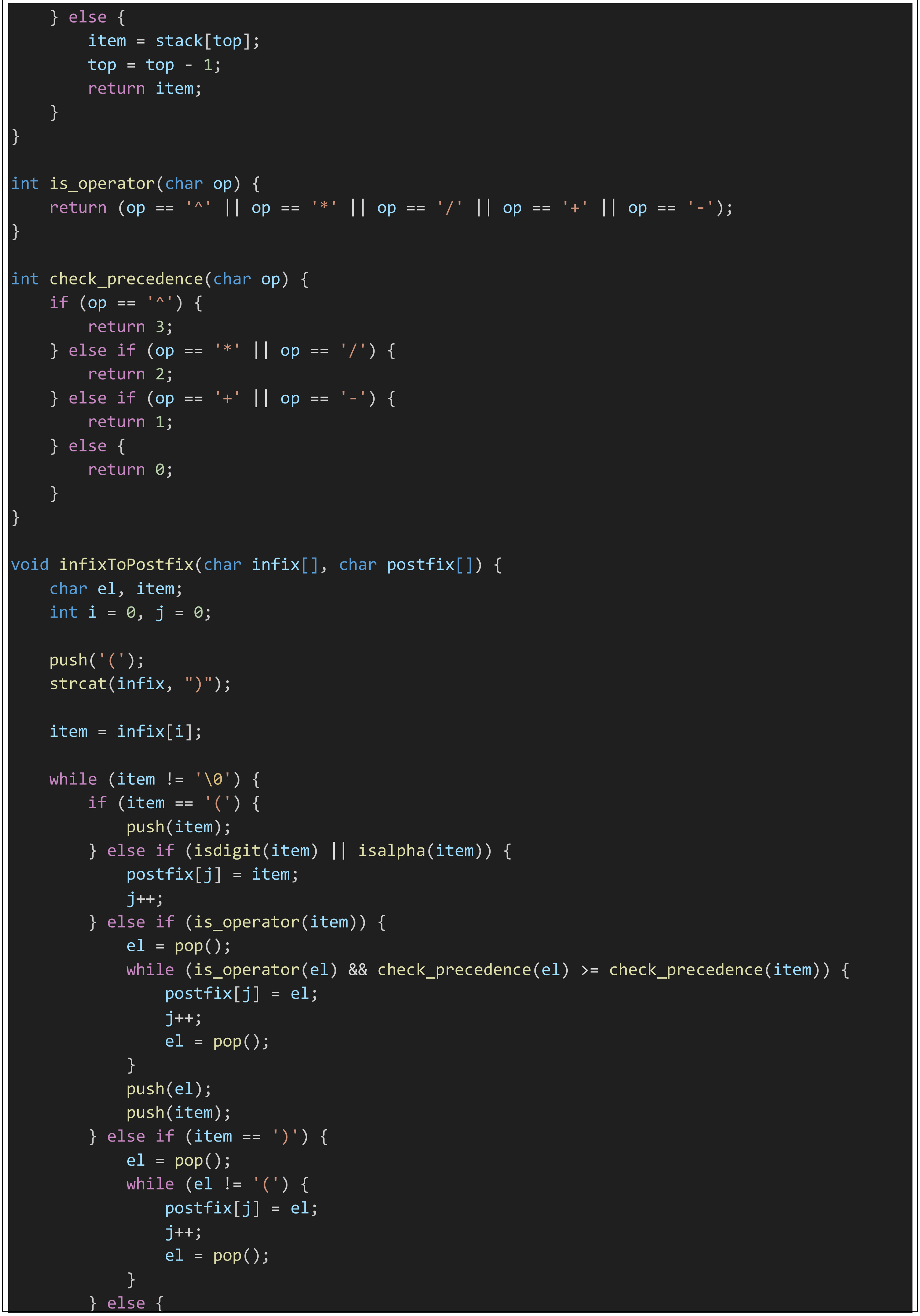
Postfix Notation: Postfix notation (also known as Reverse Polish Notation or RPN) is a mathematical notation in which operators follow their operands.

⇨ Algorithm for Infix to Postfix Conversion:

- Initialize an empty stack to store operators.
- Scan the infix expression from left to right.
- If the current token is an operand (operand can be a number or a variable), append it to the postfix expression.
- If the current token is an open parenthesis '(', push it onto the stack.
- If the current token is a closing parenthesis ')', pop operators from the stack and append them to the postfix expression until an open parenthesis '(' is encountered. Pop and discard the open parenthesis.
- If the current token is an operator, pop operators from the stack and append them to the postfix expression while they have equal or higher precedence than the current operator. Push the current operator onto the stack.
- After scanning the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression.

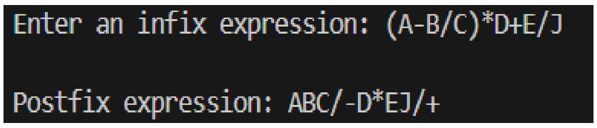
⇨ Code:

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

#define SIZE 100

int top = -1;
char stack[SIZE];

void push(char item) {
    if (top >= SIZE - 1) {
        printf("\nStack Overflow.\n");
        exit(EXIT_FAILURE);
    } else {
        top = top + 1;
        stack[top] = item;
    }
}

char pop() {
    char item;

    if (top < 0) {
        printf("\nStack Underflow!\n");
        exit(EXIT_FAILURE);
```

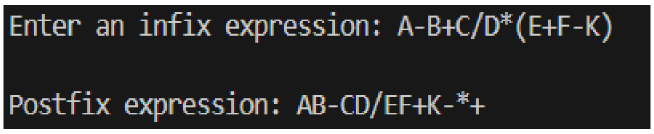


```c
    } else {
        item = stack[top];
        top = top - 1;
        return item;
    }
}

int is_operator(char op) {
    return (op == '^' || op == '*' || op == '/' || op == '+' || op == '-');
}

int check_precedence(char op) {
    if (op == '^') {
        return 3;
    } else if (op == '*' || op == '/') {
        return 2;
    } else if (op == '+' || op == '-') {
        return 1;
    } else {
        return 0;
    }
}

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

    push('(');
    strcat(infix, ")");

    item = infix[i];

    while (item != '\0') {
        if (item == '(') {
            push(item);
        } else if (isdigit(item) || isalpha(item)) {
            postfix[j] = item;
            j++;
        } else if (is_operator(item)) {
            el = pop();
            while (is_operator(el) && check_precedence(el) >= check_precedence(item)) {
                postfix[j] = el;
                j++;
                el = pop();
            }
            push(el);
            push(item);
        } else if (item == ')') {
            el = pop();
            while (el != '(') {
                postfix[j] = el;
                j++;
                el = pop();
            }
        } else {
            printf("\nInvalid expression. Exiting.\n");
            exit(EXIT_FAILURE);
        }

        i++;
        item = infix[i];
    }

    while (top >= 0) {
        el = pop();
        if (el != '(') {
            postfix[j] = el;
            j++;
        }
    }

    postfix[j] = '\0';
}

int main() {
    char infix[SIZE], postfix[SIZE];

    printf("Enter an infix expression: ");
    fgets(infix, SIZE, stdin);

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

    infixToPostfix(infix, postfix);

    printf("\nPostfix expression: %s\n", postfix);

    return 0;
}
```

⇨ **Output:**



```
Enter an infix expression: (A-B/C)*D+E/J

Postfix expression: ABC/-D*EJ/+
```



```
Enter an infix expression: A-B+C/D*(E+F-K)

Postfix expression: AB-CD/EF+K-*+
```

⇨ **Complexity:**

- **Time Complexity:** O(n), where n is the length of the infix expression. Each character is processed once.
- **Space Complexity:** O(n), where n is the length of the infix expression. The stack is used to store operators.

Assignment 6

Tower of Hanoi:

The Tower of Hanoi is a classic problem in computer science and mathematics that involves the recursive solving of a puzzle. The puzzle consists of three rods and a number of disks of different sizes that can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, and the goal is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the rods and placing it on top of another rod or on an empty rod.
3. No disk may be placed on top of a smaller disk.

⇨ **Algorithm:**

The Tower of Hanoi problem can be solved using a recursive algorithm. The key idea is to break down the problem into smaller sub-problems:

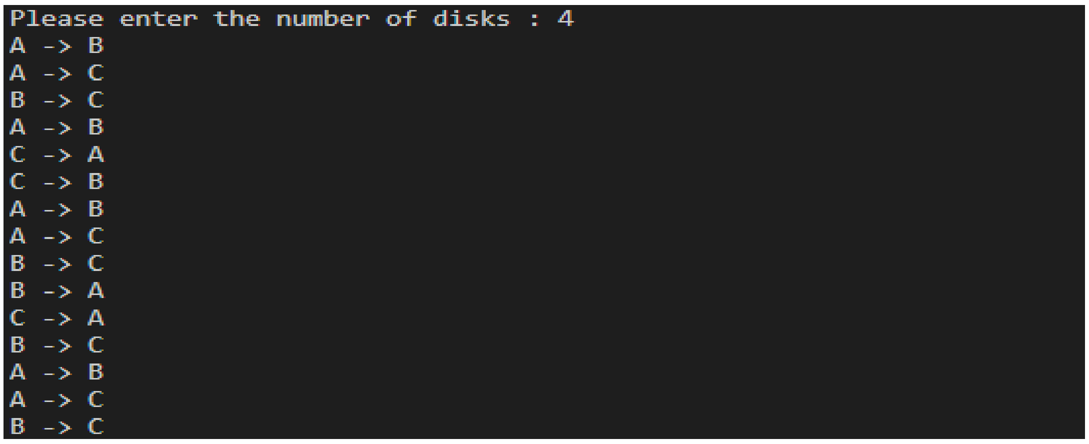
1. **Base Case:**
    - If there is only one disk, move it directly from the source rod to the destination rod.

2. **Recursive Step:**
    - Move n-1 disks from the source rod to an auxiliary rod (using the destination rod as a temporary rod).
    - Move the remaining disk from the source rod to the destination rod.
    - Move the n-1 disks from the auxiliary rod to the destination rod (using the source rod).

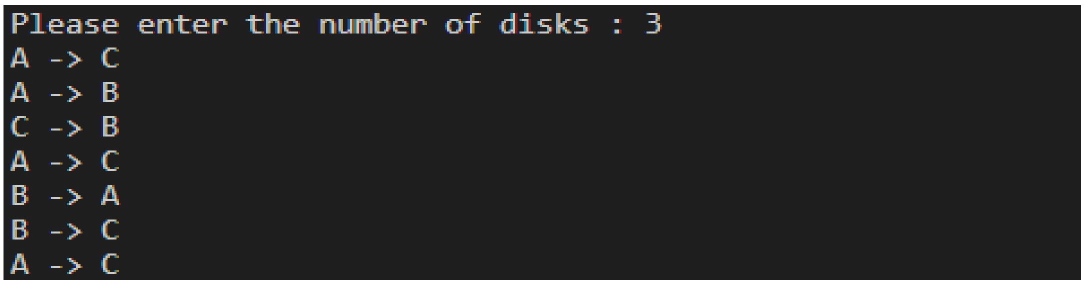
⇨ **Code:**

```c
#include <stdio.h>

void tower(int n, char s, char h, char d) {
    if (n == 0) {
        return;
    }

    tower(n - 1, s, d, h);
    printf("%c -> %c\n", s, d);
    tower(n - 1, h, s, d);
    return;
}

int main() {
    int n;

    printf("Please enter the number of disks: ");
    scanf("%d", &n);

    tower(n, 'A', 'B', 'C');

    return 0;
}
```

⇨ **Output:**



```
Please enter the number of disks : 4
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
A -> C
B -> C
B -> A
C -> A
B -> C
A -> B
A -> C
B -> C
```



```
Please enter the number of disks : 3
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

⇨ **Complexity:**

- The time complexity of the Tower of Hanoi problem is O(2^n), where n is the number of disks. This is an exponential time complexity.

⇨ **Compilation and Execution:**

1. Run **"gcc filename.c"** to compile.

2. Run **"./a.exe"** on windows or **"./a.out"** on linux.

Assignment 7
Binary Tree Traversal

A binary tree is a hierarchical data structure that consists of nodes connected by edges. Each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root. Nodes with no children are called leaves.

Basic Components:

1. **Node:**
    - Each node in a binary tree contains data and references (pointers) to its left and right children.
2. **Root:**
    - The topmost node in the tree.
3. **Leaf:**
    - A node with no children.
4. **Parent:**
    - A node that has one or more children.
5. **Child:**
    - A node directly connected to another node when moving away from the root.

⇨ **Binary Tree Traversals:**

Traversing a binary tree means visiting each node in a specific order. There are three common ways to traverse a binary tree:

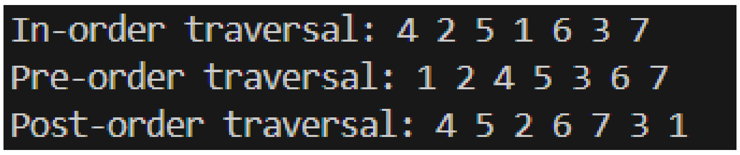
1. **Inorder Traversal (Left-Root-Right):**
    - Traverse the left subtree.
    - Visit the root node.
    - Traverse the right subtree.
2. **Preorder Traversal (Root-Left-Right):**
    - Visit the root node.
    - Traverse the left subtree.
    - Traverse the right subtree.
3. **Postorder Traversal (Left-Right-Root):**
    - Traverse the left subtree.
    - Traverse the right subtree.
    - Visit the root node.

➪ **Code:**

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

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *createNode(int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        perror("Error creating node");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

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

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

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

void freeBinaryTree(struct Node *root) {
    if (root != NULL) {
        freeBinaryTree(root->left);
        freeBinaryTree(root->right);
        free(root);
    }
}
```

```c
int main() {
    struct Node *root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);

    printf("In-order traversal: ");
    inOrderTraversal(root);
    printf("\n");

    printf("Pre-order traversal: ");
    preOrderTraversal(root);
    printf("\n");

    printf("Post-order traversal: ");
    postOrderTraversal(root);
    printf("\n");

    freeBinaryTree(root);

    return 0;
}
```

⇨ **Output:**



```
In-order traversal: 4 2 5 1 6 3 7
Pre-order traversal: 1 2 4 5 3 6 7
Post-order traversal: 4 5 2 6 7 3 1
```

⇨ **Complexity Analysis:**

- **Inorder, Preorder, and Postorder Traversal:**
  - Time Complexity: O(n)
    - Each node is visited once, where n is the number of nodes in the tree.
  - Space Complexity: O(n)
    - In the worst case, where the tree is skewed (each node has only one child), the space complexity is O(n). However, for a balanced binary tree, the space complexity is O(log n), where h is the height of the tree.

Assignment 8
Binary Search Tree Implementation

A Binary Search Tree (BST) is a binary tree data structure in which each node has at most two children, referred to as the left child and the right child. In a BST, the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree contains only nodes with keys greater than the node's key.

⇨ **Tree Operations:**

- **Insertion Operation:**
- The insert function inserts a new key into the BST while maintaining the binary search tree property.

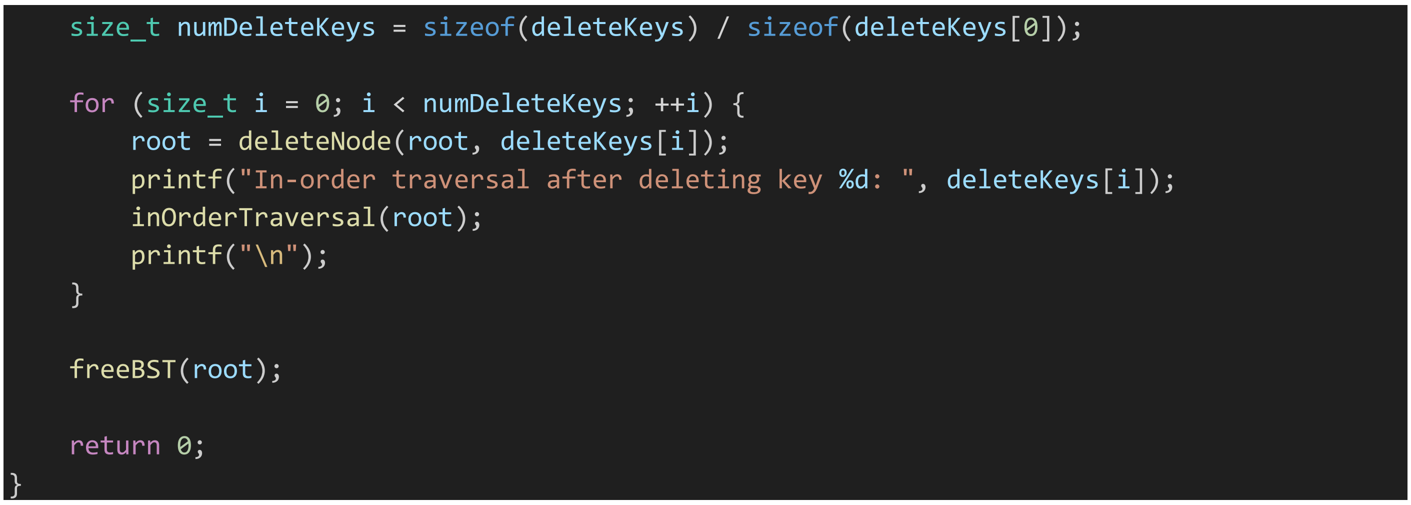
- **Deletion Operation:**
- The deleteNode function deletes a node with the given key from the BST while preserving the BST properties.

- **Search Operation:**
- The search function searches for a key in the BST and returns the node if found, or NULL otherwise.

⇨ **Code:**

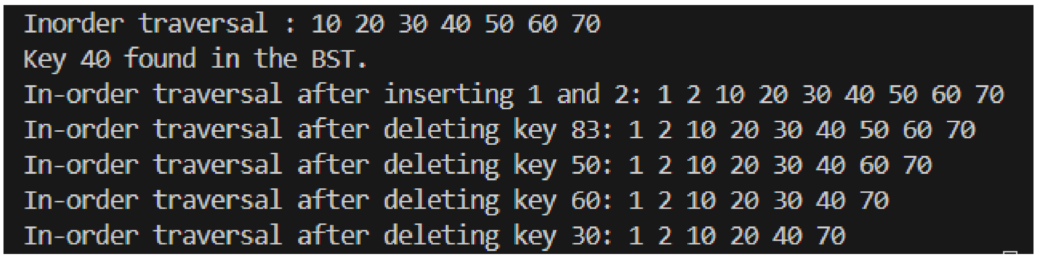
```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *createNode(int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        perror("Error creating node");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

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

struct Node *search(struct Node *root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }

    if (key < root->data) {
        return search(root->left, key);
    } else {
        return search(root->right, key);
    }
}

struct Node *insert(struct Node *root, int key) {
    if (root == NULL) {
        return createNode(key);
    }

    if (key < root->data) {
        root->left = insert(root->left, key);
    } else if (key > root->data) {
        root->right = insert(root->right, key);
    }

    return root;
}

struct Node *findMin(struct Node *node) {
    while (node->left != NULL) {
        node = node->left;
    }
    return node;
}

struct Node *deleteNode(struct Node *root, int key) {
    if (root == NULL) {
        return root;
    }

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } 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 = findMin(root->right);

        root->data = temp->data;

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

    return root;
}

void freeBST(struct Node *root) {
    if (root != NULL) {
        freeBST(root->left);
        freeBST(root->right);
        free(root);
    }
}

int main() {
    struct Node *root = createNode(40);
    root = insert(root, 30);
    root = insert(root, 50);
    root = insert(root, 20);
    root = insert(root, 60);
    root = insert(root, 10);
    root = insert(root, 70);

    printf("Inorder traversal : ");
    inOrderTraversal(root);
    printf("\n");

    int searchKey = 40;
    struct Node *searchResult = search(root, searchKey);
    if (searchResult != NULL) {
        printf("Key %d found in the BST.\n", searchKey);
    } else {
        printf("Key %d not found in the BST.\n", searchKey);
    }

    root = insert(root, 1);
    root = insert(root, 2);

    printf("In-order traversal after inserting 1 and 2: ");
    inOrderTraversal(root);
    printf("\n");

    int deleteKeys[] = {83, 50, 60, 30};
```



```c
    size_t numDeleteKeys = sizeof(deleteKeys) / sizeof(deleteKeys[0]);

    for (size_t i = 0; i < numDeleteKeys; ++i) {
        root = deleteNode(root, deleteKeys[i]);
        printf("In-order traversal after deleting key %d: ", deleteKeys[i]);
        inOrderTraversal(root);
        printf("\n");
    }

    freeBST(root);

    return 0;
}
```

⇨ **Output:**



```
Inorder traversal : 10 20 30 40 50 60 70
Key 40 found in the BST.
In-order traversal after inserting 1 and 2: 1 2 10 20 30 40 50 60 70
In-order traversal after deleting key 83: 1 2 10 20 30 40 50 60 70
In-order traversal after deleting key 50: 1 2 10 20 30 40 60 70
In-order traversal after deleting key 60: 1 2 10 20 30 40 70
In-order traversal after deleting key 30: 1 2 10 20 40 70
```

⇨ **Time Complexity Analysis:**

- **Insertion Operation:**
    - Average Case: O(log n)
    - Worst Case: O(n) (for skewed trees)
    - Space Complexity: O(log n) (recursive stack space)
- **Deletion Operation:**
    - Average Case: O(log n)
    - Worst Case: O(n) (for skewed trees)
    - Space Complexity: O(log n) (recursive stack space)
- **Search Operation:**
    - Average Case: O(log n)
    - Worst Case: O(n) (for skewed trees)
    - Space Complexity: O(log n) (recursive stack space)

Assignment 9
Graph Traversal

A graph is a data structure that consists of a set of nodes (vertices) and a set of edges connecting these nodes. Graphs are widely used to model relationships and connections between entities. They can be classified into two main types: directed graphs (digraphs), where edges have a direction, and undirected graphs, where edges have no direction.

⇨ **Basic Components of a Graph:**

1. **Vertex (Node):** Represents an entity in the graph.
2. **Edge:** Connects two vertices and may have a direction (in directed graphs).

⇨ **Types of Graphs:**

1. **Directed Graph (Digraph):** Edges have a direction from one vertex to another.
2. **Undirected Graph:** Edges have no direction; they simply connect vertices.
3. **Weighted Graph:** Assigns a weight to each edge to represent a cost or distance.
4. **Cyclic Graph:** Contains at least one cycle (a closed path).
5. **Acyclic Graph:** Does not contain any cycles.

⇨ **Complexity Analysis:**

⇨ **Adjacency Matrix Representation:**

1. **Space Complexity:** O(V^2), where V is the number of vertices.
2. **Adding/Removing Vertex:** O(V^2).
3. **Adding/Removing Edge:** O(1).
4. **Checking Edge Existence**: O(1).
5. **Traversal (DFS/BFS):** O(V^2) - Adjacency matrix may visit all vertices when searching.

⇨ **Adjacency List Representation:**

1. **Space Complexity:** O(V + E), where V is the number of vertices and E is the number of edges.
2. **Adding/Removing Vertex:** O(V + E).
3. **Adding/Removing Edge**: O(1).
4. **Checking Edge Existence**: O(V).
5. **Traversal (DFS/BFS):** O(V + E).

⇨ **Graph Algorithms:**

1. **Depth-First Search (DFS**): O(V + E).
2. **Breadth-First Search (BFS**): O(V + E).
3. **Shortest Path (Dijkstra's Algorithm):** O((V + E) * log(V)).
4. **Minimum Spanning Tree (Prim's Algorithm):** O(E * log(V)).
5. **Topological Sort:** O(V + E).

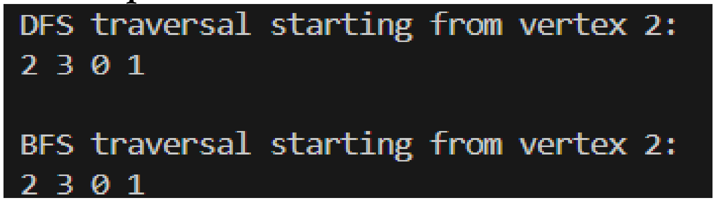
⇨ **Code:**

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

```c
#include <limits.h>

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

struct Queue {
    struct Node *front, *rear;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        perror("Error creating node");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    if (queue == NULL) {
        perror("Error creating queue");
        exit(EXIT_FAILURE);
    }
    queue->front = queue->rear = NULL;
    return queue;
}

void enqueue(struct Queue* queue, int data) {
    struct Node* newNode = createNode(data);
    if (queue->rear == NULL) {
        queue->front = queue->rear = newNode;
        return;
    }
    queue->rear->next = newNode;
    queue->rear = newNode;
}

int dequeue(struct Queue* queue) {
    if (queue->front == NULL) {
        return INT_MIN;
    }
    struct Node* temp = queue->front;
    int data = temp->data;
    queue->front = temp->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
    return data;
```

```c
}

struct Graph {
    int vertices;
    struct Node** adjList;
    bool* visited;
};

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    if (graph == NULL) {
        perror("Error creating graph");
        exit(EXIT_FAILURE);
    }
    graph->vertices = vertices;
    graph->adjList = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->visited = (bool*)malloc(vertices * sizeof(bool));
    if (graph->adjList == NULL || graph->visited == NULL) {
        perror("Error creating adjacency list or visited array");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < vertices; ++i) {
        graph->adjList[i] = NULL;
        graph->visited[i] = false;
    }
    return graph;
}

void addEdge(struct Graph* graph, int u, int v) {
    struct Node* newNode = createNode(v);
    newNode->next = graph->adjList[u];
    graph->adjList[u] = newNode;

    newNode = createNode(u);
    newNode->next = graph->adjList[v];
    graph->adjList[v] = newNode;
}

void dfsUtil(struct Graph* graph, int v) {
    graph->visited[v] = true;
    printf("%d ", v);

    struct Node* temp = graph->adjList[v];
    while (temp != NULL) {
        int neighbor = temp->data;
        if (!graph->visited[neighbor]) {
            dfsUtil(graph, neighbor);
        }
        temp = temp->next;
    }
}

void dfs(struct Graph* graph, int start) {
    for (int i = 0; i < graph->vertices; ++i) {
        graph->visited[i] = false;
    }
    dfsUtil(graph, start);
}

void bfs(struct Graph* graph, int start) {
    for (int i = 0; i < graph->vertices; ++i) {
        graph->visited[i] = false;
    }

    struct Queue* queue = createQueue();
    graph->visited[start] = true;
    printf("%d ", start);
    enqueue(queue, start);

    while (queue->front != NULL) {
        int current = dequeue(queue);

        struct Node* temp = graph->adjList[current];
        while (temp != NULL) {
            int neighbor = temp->data;
            if (!graph->visited[neighbor]) {
                graph->visited[neighbor] = true;
                printf("%d ", neighbor);
                enqueue(queue, neighbor);
            }
            temp = temp->next;
        }
    }
}

int main() {
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 0);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 3);

    printf("DFS traversal starting from vertex 2:\n");
    dfs(graph, 2);

    printf("\n\nBFS traversal starting from vertex 2:\n");
    bfs(graph, 2);

    return 0;
}
```

⇨ **Output:**



```
DFS traversal starting from vertex 2:
2 3 0 1

BFS traversal starting from vertex 2:
2 3 0 1
```

Assignment 10

Bubble Sort

⇨ **Introduction:**

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The pass through the list is repeated until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list.

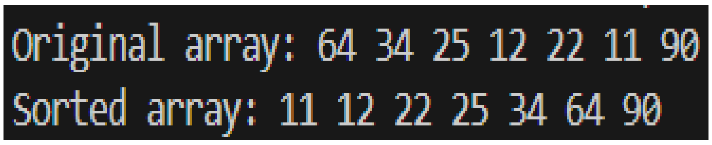
⇨ **Algorithm Steps:**

- Start with the first element (index 0) and compare it with the next element (index 1).
- If the first element is greater than the second, swap them.
- Move to the next pair of elements (index 1 and index 2) and repeat the comparison and swap if necessary.
- Continue this process until the end of the list is reached.
- After the first pass, the largest element is guaranteed to be at the end of the list.
- Repeat the process for the remaining elements, excluding the already sorted ones.
- Continue these passes until the entire list is sorted.

⇨ **Code:**

```c
#include <stdio.h>

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

void bubbleSort(int arr[], int n) {
    int swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = 0;

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
                swapped = 1;
            }
        }


        if (swapped == 0) {
            break;
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);


    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

⇨ **Output:**



```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

⇨ **Complexity Analysis:**

- **Time Complexity:**

1. Best Case: O(n) - when the list is already sorted.
2. Worst Case: O(n^2) - when the list is in reverse order.
3. Average Case: O(n^2)

- **Space Complexity:**

1. O(1), Bubble Sort is an in-place sorting algorithm, meaning it doesn't require additional memory.

Selection Sort

⇨ Introduction:

- Selection Sort is a simple sorting algorithm that repeatedly selects the minimum element from the unsorted portion of the array and swaps it with the first unsorted element.
- The algorithm divides the array into two parts: the sorted part at the beginning and the unsorted part at the end. In each iteration, the minimum element from the unsorted part is found and placed at the end of the sorted part.

⇨ Algorithm Steps:

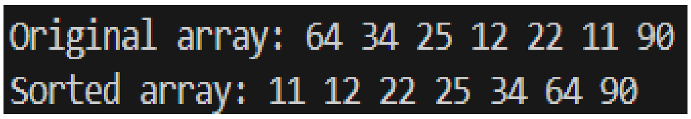
- Find the minimum element in the unsorted part of the array.
- Swap the minimum element with the first element in the unsorted part.
- Expand the sorted part to include the newly placed minimum element.
- Repeat steps 1-3 until the entire array is sorted.

⇨ Code:

```c
#include <stdio.h>

void swap(int a[], int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

void selectionSort(int arr[], int n) {
    int minIndex;

    for (int i = 0; i < n - 1; i++) {

        minIndex = i;


        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }


        swap(arr, i, minIndex);
    }
}
```

```c
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);


    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

⇨ **Output:**



```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

⇨ **Complexity Analysis:**

- **Time Complexity:**

1. Worst Case: O(n^2)
2. Best Case: O(n^2)
3. Average Case: O(n^2)

- **Space Complexity:**

1. O(1) (in-place sorting)

Insertion Sort

- Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.
- However, it performs well for small datasets or partially sorted datasets. The algorithm works by repeatedly taking one element from the unsorted part of the array and inserting it into its correct position in the sorted part.

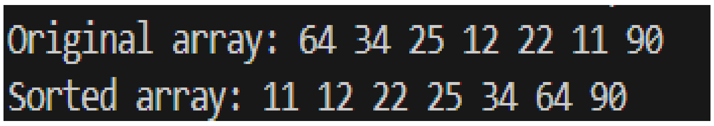
⇨ Algorithm Steps:

1. Assume the first element is sorted; start from the second element.
2. Compare the current element with the elements in the sorted part.
3. Move the greater elements to the right to make space for the current element.
4. Insert the current element in its correct position in the sorted part.
5. Repeat steps 2-4 for each remaining element.

⇨ Code:

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;


        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }


        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);


    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

⇨ **Output:**



```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

⇨ **Insertion Sort Complexity:**

- **Time Complexity:**

1. Worst Case: O(n^2)
2. Best Case: O(n) when the array is already sorted.
3. Average Case: O(n^2)

- **Space Complexity:**

1. O(1) (in-place sorting)

Quick Sort

- Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer paradigm.
- The algorithm works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

Algorithm Steps:

- **Partitioning:**

1. Choose a pivot element from the array. Common choices include the first, last, or a random element.
2. Reorder the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
3. The pivot is now in its final sorted position.

- **Recursion:**

1. Recursively apply the Quick Sort algorithm to the sub-arrays on the left and right of the pivot.

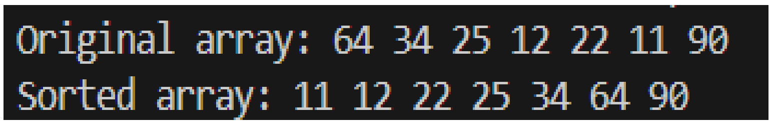
- **Base Case:**

1. The base case is when the sub-array has one or zero elements, as it is already sorted.

⇨ **Code:**

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);


    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

⇨ **Output:**



```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

⇨ **Complexity Analysis:**

- **Time Complexity:**

1. Worst Case: O(n^2) when the pivot selection consistently results in unbalanced partitions (e.g., if the array is already sorted).
2. Best Case: O(n log n) when the pivot selection consistently results in balanced partitions.
3. Average Case: O(n log n)

- **Space Complexity:**

1. Worst Case: O(log n) due to the recursive call stack.
2. Best Case: O(log n)
3. Average Case: O(log n)

Merge Sort

- Merge Sort is a comparison-based, divide-and-conquer sorting algorithm.
- It works by dividing the array into two halves, recursively sorting each half, and then merging the sorted halves to produce a fully sorted array.

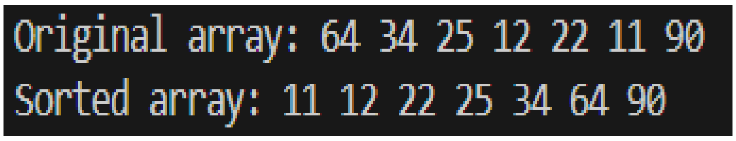
⇨ **Algorithm Steps:**

1. **Divide:**
    - Divide the unsorted array into two halves.
2. **Recursion:**
    - Recursively apply Merge Sort to each half.
3. **Conquer:**
    - Merge the sorted halves to produce a fully sorted array.

⇨ **Code:**

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

void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {

        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);


        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

⇨ **Output:**



```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

⇨ **Complexity Analysis:**

- **Time Complexity:**

1. Worst Case: O(n log n)
2. Best Case: O(n log n)
3. Average Case: O(n log n)

- **Space Complexity:**

1. Worst Case: O(n) for additional space due to the temporary array used in merging.
2. Best Case: O(n)
3. Average Case: O(n)

⇨ **Compilation and Execution:**

1. Run “**gcc filename.c**” to compile.

2. Run “**./a.exe**” on windows or “**./a.out**” on linux.

Assignment 11
Queue Implementation

A queue is a fundamental data structure that follows the First-In, First-Out (FIFO) principle. It represents a collection of elements where the first element added is the first to be removed. Queues are widely used in computer science and real-world applications, including task scheduling, print spooling, and network data packet handling.

⇨ **Operations on a Queue:**

1. **Enqueue (Insertion):** Adds an element to the rear of the queue.
2. **Dequeue (Deletion):** Removes the element from the front of the queue.
3. **Front:** Retrieves the element at the front without removing it.
4. **Rear:** Retrieves the element at the rear without removing it.
5. **IsEmpty:** Checks if the queue is empty.
6. **IsFull:** Checks if the queue is full (applies to a bounded or fixed-size queue).
7. **Size:** Returns the number of elements in the queue.
8. **Clear:** Removes all elements from the queue.

⇨ **Code:**

```c
#include <stdio.h>

#define MAX_SIZE 100

struct Queue {
    int items[MAX_SIZE];
    int front, rear;
};

void initializeQueue(struct Queue *queue) {
    queue->front = -1;
    queue->rear = -1;
}

int isEmpty(struct Queue *queue) {
    return (queue->front == -1 && queue->rear == -1);
}

int isFull(struct Queue *queue) {
    return (queue->rear == MAX_SIZE - 1);
}

void enqueue(struct Queue *queue, int value) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }

    if (isEmpty(queue)) {
        queue->front = 0;
        queue->rear = 0;
    } else {
        queue->rear++;
    }

    queue->items[queue->rear] = value;
    printf("Enqueued: %d\n", value);
}

int dequeue(struct Queue *queue) {
    int dequeuedItem;

    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    dequeuedItem = queue->items[queue->front];
    if (queue->front == queue->rear) {
        initializeQueue(queue);
    } else {
        queue->front++;
    }

    printf("Dequeued: %d\n", dequeuedItem);
    return dequeuedItem;
}

void displayQueue(struct Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Queue elements: ");
    for (int i = queue->front; i <= queue->rear; i++) {
        printf("%d ", queue->items[i]);
    }
    printf("\n");
}

int main() {
    struct Queue myQueue;
    initializeQueue(&myQueue);

    int choice, value;
    do {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&myQueue, value);
                break;

            case 2:
                dequeue(&myQueue);
                break;

            case 3:
                displayQueue(&myQueue);
                break;

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

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

    } while (choice != 4);

    return 0;
}
```

⇨ **Complexity Analysis:**

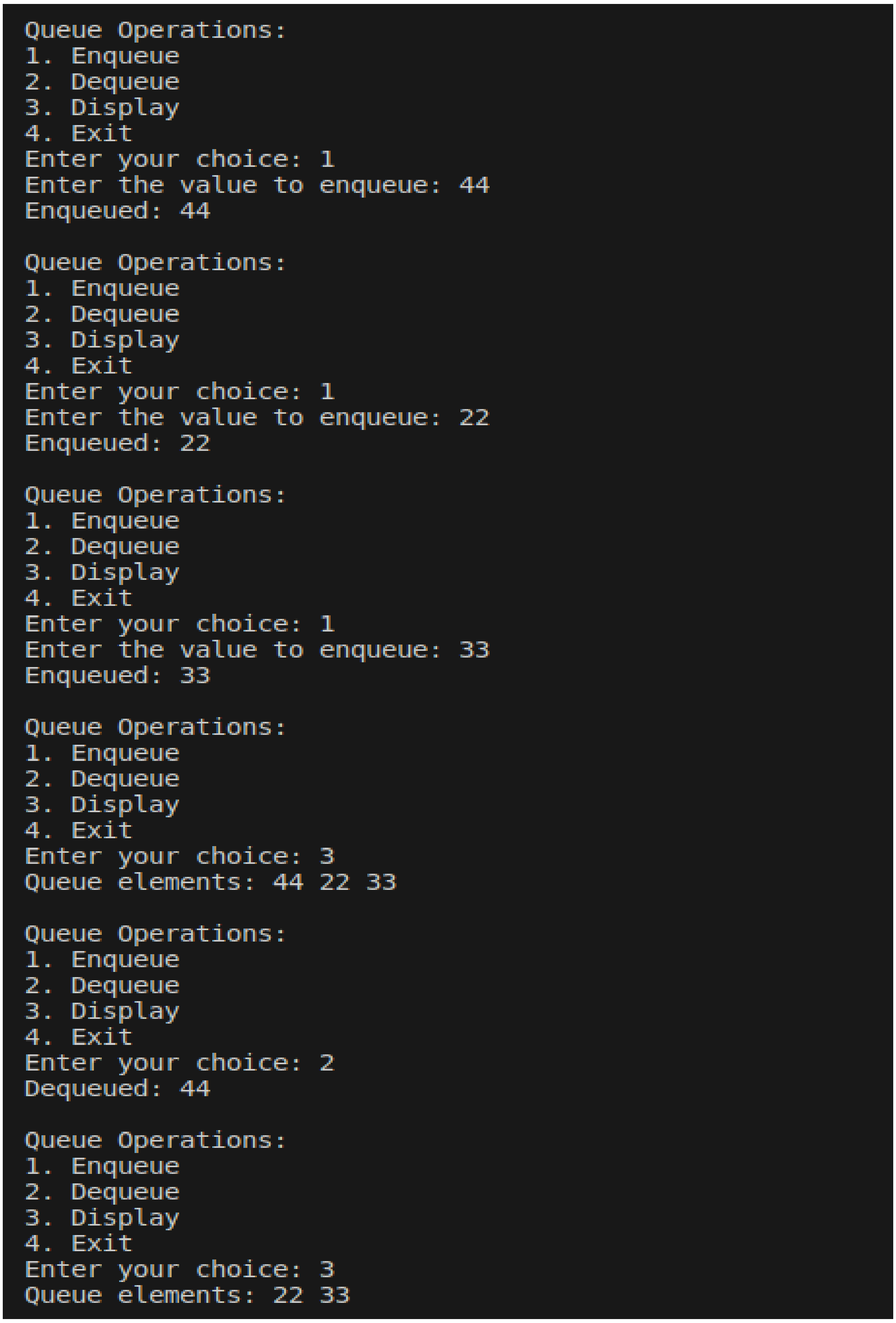
- **Time Complexity:**

1. **Enqueue: O(1) -** Constant time complexity as it involves inserting at the rear.
2. **Dequeue: O(1) -** Constant time complexity for removing from the front.
3. **Front/Rear: O(1) -** Retrieving the front or rear element takes constant time.
4. **IsEmpty: O(1)** - Checking if the queue is empty is a constant time operation.
5. **IsFull: O(1) -** For a bounded or fixed-size queue, checking if it's full is constant time.
6. **Size: O(1) -** Determining the number of elements in the queue is a constant time operation.

- ***Space Complexity:***

1. **Array-based Implementation: O(n)** - The space required is proportional to the number of elements in the queue.
2. **Linked List-based Implementation: O(n) -** The space complexity depends on the number of elements.

⇨ **Output:**



```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 44
Enqueued: 44

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 22
Enqueued: 22

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 33
Enqueued: 33

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 44 22 33

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued: 44

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 22 33
```