K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent college of Somaiya Vidyavihar University)

Name: Shreyans Tatiya

Batch: E2 Roll No.: 16010123325

Experiment No. 1

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of Abstract Data Type

Objective: Implementation of ADT without using any standard library function

Expected Outcome of Experiment:

CO	Outcome
CO 1	Explain the different data structures used in problem solving.

Books/ Journals/ Websites referred:
https://www.geeksforgeeks.org/abstract-data-types/

Abstract:-

(Define ADT. Why are they important in data structures?)

An ADT is like a conceptual contract for a data type. It defines a logical model, specifying both the behavior and the operations that can be performed on the data type.

Unlike built-in data types (such as integers or floats), ADTs are not tied to a specific memory organization or algorithm. They give us an implementation-independent view.

Here's why ADTs are important in data structures:

- **Abstraction:** ADTs hide the implementation details, allowing programmers to focus on what the data structure can do rather than how it does it.
- **Modularity and Reusability:** ADTs promote modular programming by separating the interface (what operations can be performed) from the implementation. This allows for code reuse, as the same ADT can be implemented in different ways depending on the specific needs of the program.
- **Flexibility and Efficiency:** Since ADTs are independent of implementation, different data structures (like arrays or linked lists) can be used to implement the same ADT depending on the performance requirements. This allows programmers to choose the most efficient data structure for the task at hand.
- **Maintainability:** Changes to the implementation details of an ADT don't affect the code that uses it as long as the interface remains the same. This makes code easier to maintain and update.

Abstract Data Type for Rational Numbers

[For the assigned data type, write value definition and operator definition)

Value Definition for Rational Numbers

1. **Numerator**:
    - Represents the top part of the fraction.
    - Defined as an integer.
2. **Denominator**:
    - Represents the bottom part of the fraction.
    - Defined as an integer.
    - Must not be zero to maintain a valid rational number.

Operator Definitions for Rational Numbers

1. **Create**:
    - Initializes a rational number by setting its numerator and denominator.
    - Typically involves user input to set the values.

2. **Add**:

    - Computes the sum of two rational numbers.
    - The result is a new rational number where:
    - The numerator is `(r1.numerator * r2.denominator) + (r2.numerator * r1.denominator)`.
    - The denominator is `r1.denominator * r2.denominator`.

3. **Multiply**:

    - Computes the product of two rational numbers.
    - The result is a new rational number where:
    - The numerator is `r1.numerator * r2.numerator`.
    - The denominator is `r1.denominator * r2.denominator`.

4. **Compare**:

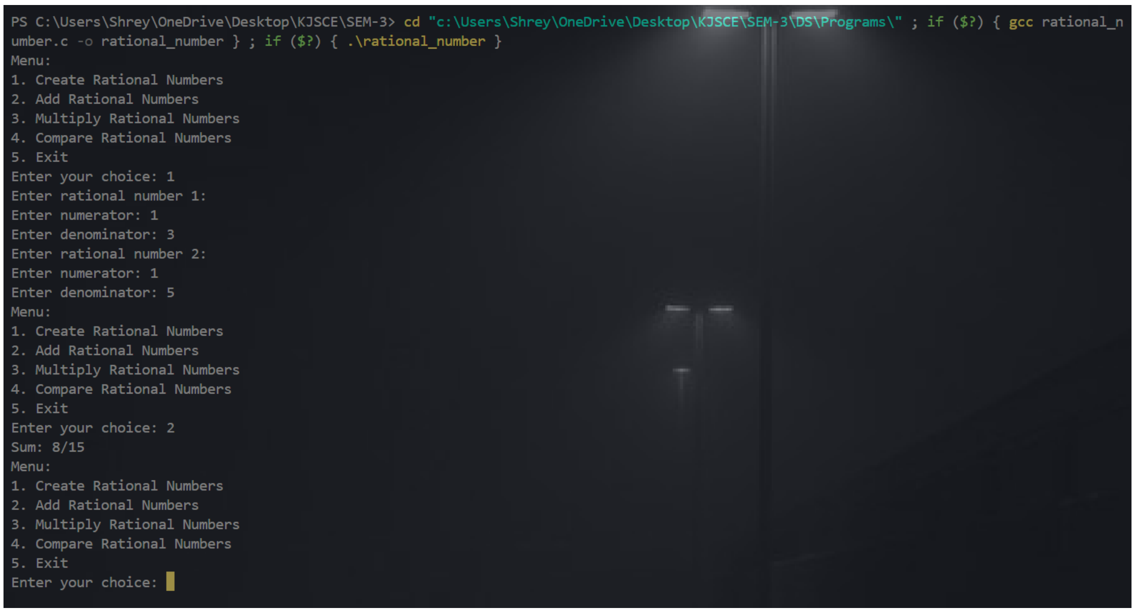
    - Compares two rational numbers to determine their relative size.
    - Possible results are:
        - `1` if the first rational number is greater.
        - `-1` if the first rational number is smaller.
        - `0` if both rational numbers are equal.
    - Comparison is done by cross-multiplying: `(r1.numerator * r2.denominator)` and `(r2.numerator * r1.denominator)`.

Implementation Details:

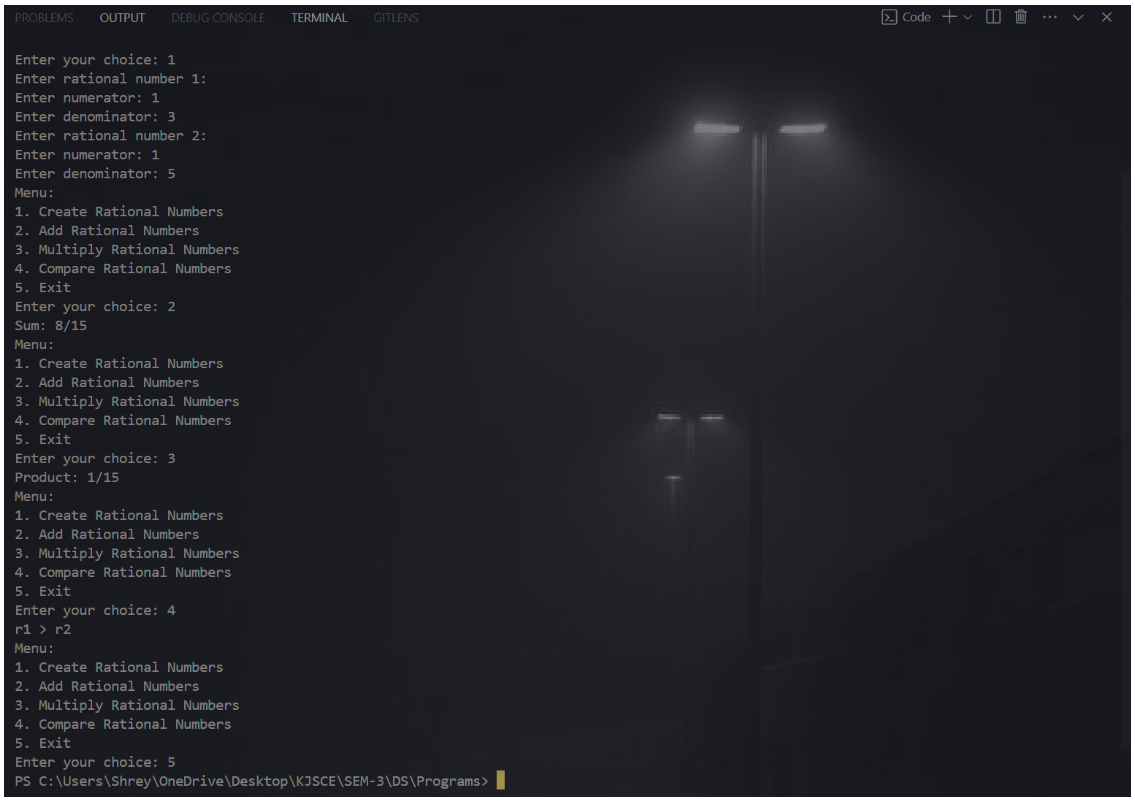
1. Enlist all the Steps followed and various options explored
2. Explain your program logic and methods used.
3. Explain the Importance of the approach followed by you

Program code and Output screenshots:

```c
#include <stdio.h>

// Structure definition for rational numbers
struct rational {
    int numerator;
    int denominator;
} rational;

// Function to create a rational number
struct rational create() {
    struct rational rational;
    printf("Enter numerator: ");
    scanf("%d", &rational.numerator);
    printf("Enter denominator: ");
    scanf("%d", &rational.denominator);
    // Check if the denominator is zero
    if (rational.denominator == 0) {
        printf("Denominator cannot be zero. Please enter a valid 
denominator.\n");
        return create(); // Recursively prompt the user until a valid 
denominator is entered
    }
    return rational;
}

// Function to add two rational numbers
struct rational add(struct rational r1, struct rational r2) {
    struct rational result;
    result.numerator = (r1.numerator * r2.denominator) + (r2.numerator * 
r1.denominator);
    result.denominator = r1.denominator * r2.denominator;
    return result;
}

// Function to multiply two rational numbers
struct rational multiply(struct rational r1, struct rational r2) {
    struct rational result;
    result.numerator = r1.numerator * r2.numerator;
    result.denominator = r1.denominator * r2.denominator;
    return result;
}

// Function to compare two rational numbers
int compare(struct rational r1, struct rational r2) {
    if ((r1.numerator * r2.denominator) > (r2.numerator *
r1.denominator)) {
        return 1; // r1 > r2
    } else if ((r1.numerator * r2.denominator) < (r2.numerator *
r1.denominator)) {
        return -1; // r1 < r2
    } else {
        return 0; // r1 == r2
    }
}

int main() {
    int c;
    struct rational r1, r2, sum, product;

    while (1) {
        // Display menu options
        printf("Menu:\n");
        printf("1. Create Rational Numbers\n");
        printf("2. Add Rational Numbers\n");
        printf("3. Multiply Rational Numbers\n");
        printf("4. Compare Rational Numbers\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &c);

        switch (c) {
            case 1:
                // Create two rational numbers
                printf("Enter rational number 1: \n");
                r1 = create();
                printf("Enter rational number 2: \n");
                r2 = create();
                break;
            case 2:
                // Add the two rational numbers
                sum = add(r1, r2);
                printf("Sum: %d/%d\n", sum.numerator, sum.denominator);
                break;
            case 3:
                // Multiply the two rational numbers
                product = multiply(r1, r2);
                printf("Product: %d/%d\n", product.numerator,
product.denominator);
                break;
            case 4:
                // Compare the two rational numbers
                if (compare(r1, r2) == 1) {
                    printf("r1 > r2\n");
                } else if (compare(r1, r2) == -1) {
                    printf("r1 < r2\n");
                } else {
                    printf("r1 = r2\n");
                }
                break;
            case 5:
                // Exit the program
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}
```



```
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3> cd "c:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS\Programs\" ; if ($?) { gcc rational_n
umber.c -o rational_number } ; if ($?) { .\rational_number }
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 1
Enter rational number 1:
Enter numerator: 1
Enter denominator: 3
Enter rational number 2:
Enter numerator: 1
Enter denominator: 5
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 2
Sum: 8/15
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 
```



```
Enter your choice: 1
Enter rational number 1:
Enter numerator: 1
Enter denominator: 3
Enter rational number 2:
Enter numerator: 1
Enter denominator: 5
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 2
Sum: 8/15
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 3
Product: 1/15
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 4
r1 > r2
Menu:
1. Create Rational Numbers
2. Add Rational Numbers
3. Multiply Rational Numbers
4. Compare Rational Numbers
5. Exit
Enter your choice: 5
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS\Programs> 
```

Conclusion:-

We implemented the ADT for rational numbers without using any standard library functions, providing custom operations for creation, addition, multiplication, and comparison.

Post lab questions:

1) Discuss the Advantages of Abstract Data Types.

ANS:

Advantages:

Encapsulation: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.

Abstraction: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.

Data Structure Independence: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.

Information Hiding: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.

Modularity: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming

2) Compare and Contrast between ADT, Data Types and Data Structure.

Aspect	**Abstract Data Types (ADTs)**	**Data Types**	**Data Structures**
Definition	Concept defining operations and properties	Classification of data values	Concrete implementation of ADTs
Focus	Operations and properties	Value classification and operations	Data storage and organization
Implementation	Abstract, not specified	Language-specific (primitive or composite)	Concrete, specified in code
Examples	List, Queue, Stack	int, float, array, struct	Array, Linked List, Tree, Hash Table
Encapsulation	Encapsulates data and operations	May or may not encapsulate	Can encapsulate or be exposed

Efficiency Concerns	Not concerned with implementation efficiency	Depends on the type	Highly concerned with efficiency

3) Define 5 ADT functions for Lists.

Ans:

- **isEmpty(list):** This function checks if the list is empty. It takes the list as input and returns a boolean value (True if empty, False otherwise).
- **size(list):** This function returns the number of elements in the list. It takes the list as input and returns an integer value representing the size.
- **get(list, index):** This function retrieves the element at a specific position (index) within the list. It takes the list and the index as input and returns the element at that index. **Precondition:** The index must be within the valid range of the list (0 to size-1).
- **insert(list, element, index):** This function inserts a new element into the list at a specified position (index). It takes the list, the element to be inserted, and the index as input. It modifies the original list by inserting the element at the specified position. **Precondition:** The index must be within the valid range (0 to size for insertion at the end).
- **remove(list, element):** This function removes the first occurrence of a specific element from the list. It takes the list and the element to be removed as input and modifies the original list by removing the element (if found). It returns a boolean value indicating success (True if element was removed, False otherwise).