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

Batch: D3 Roll No.: 16010123294

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.

Abstract:-

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

An Abstract Data Type (ADT) is a mathematical model for a certain class of data structures that have similar behaviour. An ADT is defined by its behaviour (operations), not by its implementation. It specifies:

Data: The type of data that the ADT will hold.

Operations: The operations that can be performed on the data, including their behaviour and properties.

Importance:

Encapsulation and Abstraction:

ADTs encapsulate the data and the operations that can be performed on the data, hiding the implementation details. This allows programmers to work with higher-level abstractions without worrying about the underlying complexities.

Modularity:

ADTs promote modularity by allowing the separation of the interface and implementation. This makes the code easier to maintain and modify. For example, you can change the internal implementation of an ADT without affecting the code that uses it.

Reusability:

ADTs allow for code reuse. Once an ADT is defined, it can be used in multiple programs. This reduces redundancy and improves the efficiency of the development process.

Improved Readability and Maintenance:

By providing a clear and concise interface, ADTs make programs easier to read and understand. This also simplifies debugging and maintenance.

Flexibility:

Different implementations of an ADT can be created to optimize for various performance characteristics (e.g., time complexity, space complexity). This allows developers to choose or switch to the most appropriate implementation based on their needs.

Separation of Concerns:

ADTs separate the concerns of what operations are performed from how these operations are implemented. This allows developers to focus on designing the functionality before delving into the implementation specifics.

Abstract Data Type for Rational Numbers

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

Value Definition:

abstract typedef <Integer x , integer y >RationalType;

condition: y!=0

Operator Definition:

Add:

abstract RationalType

add<x,y>

integer x,y;

Pre-conditon : None

Post-Conditon :

add[0]: (x[0] * y[1]) + (y[0] * x[1])

add[1]: x[1] * y[1]

Subtract:

abstract RationalType

sub<x,y>

integer x,y;

Pre-conditon : None

Post-Conditon :

sub[0]: (x[0] * y[1]) - (y[0] * x[1])

sub[1]: x[1] * y[1]

Multiply:

abstract RationalType

mult<x,y>

Integer x,y;

Pre-conditon : None

Post-Conditon :

mult[0]: x[0] * y[0])

mult[1]: x [1] * y[1]

Equality:

abstract RationalType

equal<x,y>

Rationaltype x,y;

Pre-conditon : None

Post-Conditon :

equal = true

if (x[0] * y[1]) = = (y[0] * x[1])

Implementation Details:

1. Enlist all the Steps followed and various options explored

Created a typedef structure named rational that contains two integers: num for the numerator and den for the denominator.
Declared function prototypes for add, sub, mult, and equal which operate on two rational structures.

In the main function, declared two rational variables r1 and r2 to store the input rational numbers and one rational variable result to store the result. Asked the user to input two rational numbers in the format a/b and c/d. Implemented a loop to present a menu to the user with options to add, subtract, multiply, check equality, or exit the program. Used a switch-case structure to call the corresponding function based on the user's choice and displayed the result. Implemented the add, sub, mult, and equal functions to perform the respective operations on the rational numbers and return the result.

Various Options Explored:

The code assumes that the user enters the rational numbers in the correct format and does not handle invalid inputs or zero denominators. The current implementation does not simplify the resulting rational number (e.g., 2/4 to 1/2).

2. Explain your program logic and methods used.

Program Logic:

The program begins by defining a rational structure and declares the necessary function prototypes. The main function prompts the user to input two rational numbers and presents a menu to select an operation. Depending on the user's choice, it performs the corresponding operation using the implemented functions and displays the result.

Methods Used:

Addition (add):

To add two rational numbers a/b and c/d we use the formula
result= (a*d + c*b)/(b*d)

Subtraction (sub):

To subtract two rational numbers a/b and c/d we used the formula

result= (a*d - c*b)/(b*d)

Multiplication (mult):

To multiply two rational numbers a/b and c/d we used the formula

Result= (a*c) / (b*d)

Equality Check (equal):

To check if two rational numbers a/b and c/d we checked for Boolean value of
a*d = = b*c (if true than equal else false)

3. Explain the Importance of the approach followed by you

By defining separate functions for each operation (add, sub, mult, equal), the code follows a modular approach which makes it easy to understand, maintain, and extend. Each function handles a specific task, promoting code reuse and readability.

The program provides a menu-driven interface, allowing users to perform multiple operations in a single run. This improves user experience by offering flexibility and interactivity.

Using a typedef structure for rational numbers abstracts the details of the rational number representation from the main logic. This encapsulation ensures that the operations on rational numbers are performed consistently and correctly.

The methods used for addition, subtraction, and multiplication of rational numbers follow standard arithmetic rules, ensuring accurate results.

Program code and Output screenshots:

```
//Exp1

#include<stdio.h>

#include<stdbool.h>

typedef struct{

int num;

int den;
```

```
}rational;

rational add(rational r1,rational r2);

rational sub(rational r1,rational r2);

rational mult(rational r1,rational r2);

bool equal(rational r1,rational r2);


int main(){

rational r1,r2,result;

int choice;


printf("Enter First rational number (a/b)\n");

scanf("%d/%d",&r1.num,&r1.den);

printf("Enter Second rational number (c/d)\n");

scanf("%d/%d",&r2.num,&r2.den);

while (true)

{

printf("Operations:\n");

printf("1. Add\n");

printf("2. Sub\n");

printf("3. Multi\n");

printf("4. Equal\n");

printf("5. Exit\n");

printf("Enter choice (1-5): ");

scanf("%d", &choice);
```

```
switch (choice) {
  case 1:
     result = add(r1, r2);
     printf("Result of addition:%d/%d\n",result.num,result.den);

     break;
  case 2:
     result = sub(r1, r2);
     printf("Result of subtraction:%d/%d\n",result.num,result.den );

     break;
  case 3:
     result = mult(r1, r2);
     printf("Result of multiplication:%d/%d\n",result.num,result.den );

     break;
  case 4:
     if (equal(r1,r2))
     {
        printf("Rational numbers are equal.\n");
     }
     else
     {
        printf("Rational numbers are not equal.\n");
     }
```

```
            break;


    case 5:
            printf("Exit the program.\n");
                return 0;
    default:
                printf("Invalid choice.\n");
                break;
}
}
return 0;
}
rational add(rational r1, rational r2) {
    rational result;
    result.num = r1.num * r2.den + r2.num * r1.den;
    result.den = r1.den * r2.den;
    return result;
}
rational sub(rational r1,rational r2){
    rational result;
    result.num = r1.num * r2.den - r2.num * r2.den ;
    result.den = r1.den* r2.den;
    return result;
}
rational mult(rational r1,rational r2){
```

```
    rational result;
    result.num = r1.num * r2.num;
    result.den = r1.den * r2.den;
    return result;
}
bool equal(rational r1,rational r2){
    return (r1.num * r2.den == r2.num *r1.den);
}
```

OUTPUT

:

```
PS C:\Users\Saish\OneDrive\Desktop\javatut> gcc exp-1.c
PS C:\Users\Saish\OneDrive\Desktop\javatut> ./a.exe
Enter First rational number (a/b)
2/3
Enter Second rational number (c/d)
6/4
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 1
Result of addition:26/12
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 2
Result of subtraction:-16/12
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 3
Result of multiplication:12/12
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 4
Rational numbers are not equal.
Operations:
1. Add
2. Sub
```

```
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 4
Rational numbers are not equal.
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 6
Invalid choice.
Operations:
1. Add
2. Sub
3. Multi
4. Equal
5. Exit
Enter choice (1-5): 5
Exit the program.
```

Conclusion:-

Learned about ADT and implemented it in the program and can used in further cases as well.

Also learned about writing Value definition and Operator Definition for the functions created in the program.

Post lab questions:

1) Discuss the Advantages of Abstract Data Types.

Encapsulation : ADTs encapsulate data and the operations that can be performed on that data. This encapsulation ensures that the internal representation of the data is hidden from the user, promoting: Data Integrity: By restricting direct access to data, ADTs prevent unauthorized or accidental modifications, ensuring that the data remains consistent and valid. Simplified Interface: Users interact with the ADT through a defined set of operations, making the data type easier to use and understand.

Modularity : ADTs support modularity by enabling the separation of concerns in software development. This leads to: Independent Development: Different parts of a program can be developed and tested independently, improving the overall efficiency of the development process. Easier Maintenance: Changes in the implementation of an ADT can be made without affecting the rest of the program, as long as the interface remains consistent.

Reusability : ADTs promote code reuse by providing generic and reusable components. This helps in: Reducing Redundancy: Common data structures and operations can be defined once and reused across multiple projects. Improving Productivity: Developers can leverage existing ADTs to build new functionalities faster, rather than writing code from scratch.

Abstraction : ADTs provide a level of abstraction that allows developers to focus on what operations can be performed rather than how they are implemented. This abstraction leads to: Simplified Problem Solving: By focusing on high-level operations, developers can solve complex problems more easily without getting bogged down by low-level details. Flexibility: Different implementations of the same ADT can be used interchangeably, allowing for performance optimizations and adaptations to different use cases.

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

Abstraction Level:

ADT: High-level, focuses on what operations are performed.

Data Type: Low-level, defines the type of data and basic operations.

Data Structure: Mid-level, focuses on how data is organized and managed.

Implementation:

ADT: Implementation-independent, abstract description.

Data Type: Directly supported by programming languages.

Data Structure: Specific implementation of an ADT, often involves multiple data types.

Usage:

ADT: Used to define abstract models for problem-solving.

Data Type: Used to declare variables and define data in programs.

Data Structure: Used to implement algorithms and manage data efficiently.

3) Define 5 ADT functions for Lists.
   - Insert
     void insert(List* list, int index, ElementType element);
   - Delete
     ElementType delete(List* list, int index);
   - Find
     int find(List* list, ElementType element);
   - Get
     ElementType get(List* list, int index);
   - Size
     int size(List* list);