List 03 – Data Structures – Linear Data Structures

Arrays

1) Initialize a ten positions integer array named numbers with random integers between 10 and 90 (inclusive).
   Implement methods to:
   a) Check if 50 exists (print "Found" or "Not found")
   b) Count occurrences of 50 and print the count.
   c) Compute and print the average as a floating-point number.
   d) Find and print the maximum and minimum values.
   e) Print the sum and the product of all elements (use an appropriate numeric type to avoid overflow).
   f) Print the array in reverse order.
   g) Create another array named **reversedNumbers** with elements copied in reverse from numbers and print it.
   h) Count even and odd numbers and then create two new arrays to storage **evenNumbers** and **oddNumbers**; copy even and odd elements from **numbers** into them.

2) Read an integer n (n > 0) representing how many integers will be entered. Then read n integers into each correspondent array position. For each index i (0..n-1), print the product values[i] * i.
   Input
      Number of values: 5
      4 5 2 7 1

   Output
      Result: 0 2 14 3

3) Ask the user for the number of throws N. Simulate N throws of a fair six-sided die (values 1..6), store results in an array named throws with 6 positions, and compute the percentage frequency of each face (1..6). Print each face's count and percentage with suitable precision (e.g., two decimals).
   Input:
      Throws: 10

   Simulated output (one possible run):
      Number of throws: 10
      Face 1: 2 (20.00%)
      Face 2: 1 (10.00%)
      Face 3: 3 (30.00%)
      Face 4: 1 (10.00%)
      Face 5: 2 (20.00%)
      Face 6: 1 (10.00%)

4) Read a 4x4 matrix of integers. Then display:
    a) the sum of all elements;
    b) the sum of the elements in the second row;
    c) the sum of the elements in the third column;
    d) the elements of the main diagonal (from top-left to bottom-right);
    e) the elements of the secondary diagonal (from top-right to bottom-left).

    Input:

    16 integers representing the 4x4 matrix (row by row).

    Output:

    a) Total sum: <value>
    b) Sum of second row: <value>
    c) Sum of third column: <value>
    d) Main diagonal elements: <e00>, <e11>, <e22>, <e33>
    e) Secondary diagonal elements: <e03>, <e12>, <e21>, <e30>

5) An integer square matrix is called a **magic square** if the sum of the elements of each row, the sum of the elements of each column, and the sums of the main and secondary diagonals are all equal.
    Example (3x3): 8 0 7
    4 5 6
    3 10 2

    This is a magic square because: 8+0+7 = 4+5+6 = 3+10+2 = 8+4+3 = 0+5+10 = 7+6+2 = 8+5+2 = 3+5+7

    Given a square matrix A (n x n) of integers, verify whether A is a magic square. Print a clear result (e.g., "Magic square" or "Not a magic square").

    Input:

    Integer n (matrix size), followed by n × n integers (row-major order).

    Output:

    A single message indicating whether the matrix is a magic square.

    Notes:
    - n must be >= 1.
    - Use integer arithmetic for all sums. If needed, clarify handling of trivial cases (e.g., n = 1 is a magic square).

List, Stack, and Queue

1) Create a program that operates on a list of integers and displays the following menu:

    1) Add a number
    2) Remove a number
    3) Remove by position
    4) Print the list
    5) Print the list in reverse order
    6) Print the number of elements in the list
    7) Clear all elements from the list
    8) Exit the program

    The program should display appropriate error messages when errors occur (for example: invalid input, remove of a value or position that does not exist, out-of-range positions, etc.). The menu must repeat until the user chooses to exit.

2) Create a program that operates on a stack of integers and displays the following menu. The program should repeat the menu until the user chooses to exit. Use an appropriate stack data structure (e.g., Stack<int> in C#).

    Menu
    1) Push a number
    2) Pop a number
    3) Peek (show top)
    4) Clear the stack
    5) Exit the program

    Behavior details:

    - Push a number: prompt for an integer and push it onto the stack; confirm with "Number pushed."
    - Pop a number: remove and show the top element. If the stack is empty, display an appropriate error message (e.g., "Stack is empty. Cannot pop.").
    - Peek (show top): display the top element without removing it. If empty, show an error message.
    - Print the number of elements in the stack: display the stack size.
    - Clear the stack: remove all elements and confirm with "Stack cleared."

    Error handling:

    - Validate user input for menu choices and integer entries.
    - Show clear error messages for invalid input and operations on an empty stack.

3) Create a program that operates in a queue of **strings** and displays the following menu. The program should repeat the menu until the user chooses to exit. Use an appropriate queue data structure.

Menu

1) Enqueue a string
2) Dequeue a string
3) Peek (show front)
4) Print all elements
5) Print the number of elements in the queue
6) Clear the queue
7) Exit the program

Behavior details:

- Enqueue a string: prompt for a string and add it to the end of the queue; confirm with "String enqueued."
- Dequeue a string: remove and show the front element. If the queue is empty, display an appropriate error message (e.g., "Queue is empty. Cannot dequeue.").
- Peek (show front): display the front element without removing it. If empty, show an error message.
- Print all elements: print queue contents from front to rear.
- Print the number of elements in the queue: display the queue size.
- Clear the queue: remove all elements and confirm with "Queue cleared."
- Exit the program: terminate the loop and program.

Error handling
- Validate menu selections and integer inputs.
- For removal/peek operations on empty structures, print descriptive error messages.