Stack - Exercise

Problem 1: Reverse String

Implement a C function to reverse an input string using a stack of characters.

Function Declaration: char* reverse(char *str);

Sample:

Input: bring Output: gnirb

Problem 2: Scan Binary

Given a binary number as a string, implement a C function to scan the string from left to right, push the character on encountering 1, pop on encountering 0, and return the number of elements in the stack after completely scanning the string.

Function Declaration: int scan(char *str);

Sample:

Input: 101010101111100 Output: 3

Problem 3: Validate Parenthesis

Given a string containing different types of parentheses, implement a C function (validateParantheses) using stack to check whether each opening parenthesis has a matching closing parenthesis and the order in which the parentheses are closed is the same as their order of opening.

Function Declaration: int validateParantheses(char * string);

Sample:

Input: Equality of adjacent items (A[i] = A[i + 1]) does not occur
Output: 1 ➔ indicates true

Input: Equality of adjacent items, i.e. A[i] = A[i + 1]) does not occur
Output: 0 ➔ indicates false

Input: for (int i = 0; i < strlen(s); i+=A[i]) { A[i] = (int)s[i]; }
Output: 1

Input: for (int i = 0; i < strlen(s); i+=A[i]) { A[i] = (int)s[i); }
Output: 0

Problem 4: Infix to Postfix Expression Conversion

A linked list, having an operand/operator in each node, when interpreted in order from the beginning forms a mathematical expression in C language. Hence, the expression comprises of variables, constants, operators (+, -, *, /, ^), and parenthesis. Write a C function (infixToPostfix) using stack to check if the expression is in its valid infix form, and if yes, convert the expression to its postfix form. If otherwise, NULL should be returned by the function.

Function Declaration: struct Node * infixToPostfix(struct Node * expr);

Sample:

Input	**Mathematical Expression**	**Validity**	**Postfix Expression**	**Output**
1 -> + -> ab -> 1 -> NULL	1 + ab 1	no		NULL
1 -> + -> ab1 -> NULL	1 + ab1	yes	1 ab1 +	1 -> ab1 -> + -> NULL
1 -> + -> ab1 -> * -> NULL	1 + ab1 *	no		NULL
NULL		yes		NULL
(-> 1 -> + -> ab1 -> * -> 5 -> NULL	(1 + ab1 * 5	no		NULL
1 -> + -> ab1 ->) -> * -> 5 -> NULL	1 + ab1) * 5	no		NULL
(-> 1 -> + -> ab1 ->) -> * -> 5 -> NULL	(1 + ab1) * 5	yes	1 ab1 + 5 *	1 -> ab1 -> + -> 5 -> * -> NULL
1 -> + -> ab1 -> * -> 5 -> NULL	1 + ab1 * 5	yes	1 ab1 5 * +	1 -> ab1 -> 5 -> * -> + -> NULL
7 -> + -> 5 -> * -> 3 -> / -> 5 -> ^ -> 1 -> + -> (-> 3 -> – -> 2 ->) -> NULL	7 + 5 * 3 / 5 ^ 1 + (3 - 2)	yes	7 5 3 * 5 1 ^ / + 3 2 - +	7 -> 5 -> 3 -> * -> 5 -> 1 -> ^ -> / -> + -> 3 -> 2 -> - -> + -> NULL
8 -> * -> (-> 5 -> ^ -> 4 -> + -> 2 ->) -> - -> 6 -> ^ -> 2 -> / -> (-> 9 -> * -> 3 ->) -> NULL	8 * (5 ^ 4 + 2) - 6 ^ 2 / (9 * 3)	yes	8 5 4 ^ 2 + * 6 2 ^ 9 3 * / -	8 -> 5 -> 4 -> ^ -> 2 -> + -> * -> 6 -> 2 -> ^ -> 9 -> 3 -> * -> / -> - -> NULL
4 -> ^ -> 2 -> ^ -> 3 -> NULL	4 ^ 2 ^ 3	yes	4 2 3 ^ ^	4 -> 2 -> 3 -> ^ -> ^ -> NULL
4 -> + -> 2 -> + -> 3 -> NULL	4 + 2 + 3	yes	4 2 + 3 +	4 -> 2 -> + -> 3 -> + -> NULL
4 -> / -> 2 -> * -> 3 -> NULL	4 / 2 * 3	yes	4 2 / 3 *	4 -> 2 -> / -> 3 -> * -> NULL
As -> NULL	As	yes	As	As -> NULL
5 -> NULL	5	yes	5	5 -> NULL
a -> b -> (-> c -> - -> d ->) -> NULL	a + b (c - d)	no		NULL
a -> + -> (-> c -> - -> d ->) -> b -> NULL	a + (c - d) b	no		NULL

Problem 5: Three in One

Given an array of size n, implement 3 stacks (*stack 1, stack 2* and *stack 3*) in the array, i.e the following functions are to be implemented:

```
int push(int array[], int n, int s, int data);
        // return 1 on successful push into the respective stack and 0 when unable to push.
int pop(int array[], int n, int s);
int isFull(int array[], int n, int s);
int isEmpty(int array[], int n, int s);
int peek(int array[], int n, int s);
```

In all these functions, the input variable 's' indicates that the respective operation is to be performed on *stack* <*s*>. For simplicity, initially start implementing stacks with fixed length and as a follow-up implement stacks of variable length.