Assignment no. 2

Title: Stack

Part 1 and Part 2 are mandatory.
Part 3 is bonus. You can expect part 3 type questions in the end semester exam.

- Part 1
    1. Stack Implementation:
        a. Define a constant "MAX_SIZE" to represent the maximum size of the stack.
        b. Define an integer array named "stack" with a size of "MAX_SIZE" to represent the stack.
        c. Declare an integer variable named "top" and initialize it to -1. This variable will track the index of the topmost element in the stack.
        d. Create functions for the following stack operations:
            - "push" - adds an element to the top of the stack.
            - "pop" - removes the topmost element from the stack.
            - "isEmpty" - checks if the stack is empty.
            - "isFull" - checks if the stack is full.
            - "display" - prints the elements of the stack.
        e. Implement the functions mentioned above to perform the respective operations.
    2. Stack Operations:
        a. Create an empty stack using the defined stack implementation.
        b. Push some integer values onto the stack.
        c. Print the elements of the stack using the "display" function.
        d. Pop elements from the stack and print the popped elements.
        e. Check if the stack is empty and print the result.
        f. Push additional elements onto the stack and print the updated stack.

- Part 2
    1. Parentheses Matching:
        a. Implement a function named "isParenthesesMatch" that takes a string as a parameter and checks if the parentheses in the string are balanced.
        b. The function should use a stack to check for balanced parentheses.
        c. Test the function by passing strings with balanced and unbalanced parentheses and print the results.
    2. Repeat the above assignment (1-2 in Part 1 and 1 in Part 2) using linked lists as a replacement for array.

- Part 3
    1. Expression Evaluation: Implement a function that takes a mathematical expression in infix notation as a string and evaluates it using the stack-based approach for infix to postfix conversion and postfix evaluation. Test the function by passing different mathematical expressions and printing the results.
    2. Stack Reversal: Implement a function that takes a stack as a parameter and reverses its order using an auxiliary stack. Print the elements of the reversed stack to verify the reversal.
    3. Stack-based Algorithm: Implement a stack-based algorithm such as postfix expression evaluation, infix to postfix conversion, or infix to prefix conversion. Choose one algorithm and implement it using the stack data structure. Test the function by passing suitable inputs and printing the results.
    4. Function Call Stack: The stack is extensively used in programming languages to handle function calls and manage local variables. When a function is called, its execution context, including parameters and local variables, is pushed onto the stack. When the function completes execution, its context is popped from the stack, and control is returned to the calling function. This mechanism allows for nested function calls and efficient memory management.
    5. Undo/Redo Operations: Stacks are used to implement the undo/redo functionality in various applications, including text editors, graphic design software, and command-line interfaces. Each performed action is stored on a stack, allowing the user to undo the last action by popping it from the stack. The undone actions can be pushed onto a redo stack, enabling the user to redo them if needed.
    6. Browser History: Web browsers utilize stacks to maintain the history of visited web pages. Each time a user visits a new page, the URL is pushed onto the stack. When the user clicks the "Back" button, the last visited URL is popped from the stack, allowing navigation to the previous page. Similarly, the "Forward" button can be implemented using a separate stack to store URLs when the user navigates back.
    7. Balancing Symbols: Stacks are helpful in checking the balance of symbols, such as parentheses, braces, and brackets, in programming languages. As code is parsed, opening symbols are pushed onto the stack, and when a closing symbol is encountered, it is compared with the topmost symbol on the stack. If they match, the opening symbol is popped, indicating balanced symbols. Any imbalance in the stack indicates an error in the code.
    8. Backtracking Algorithms: Various algorithms, like depth-first search (DFS) and backtracking, employ stacks to keep track of the visited nodes or states. In DFS, the stack is used to store unvisited adjacent nodes during traversal, allowing for backtracking to explore other paths. Similarly, backtracking algorithms store the state of the search space on a stack, enabling exploration of alternative paths when needed.