Assignment Report

Course: MSML606-PWS1: Algorithms and Data Structures for Machine Learning

Student Name: Ateeq Ur Rehman

Q1: Valid Parentheses Checker

- **My Approach:**
  I solved this problem using a stack. First, I started by creating an empty stack. Then I read each bracket in the given string from left to right. When I saw an opening bracket ((, [, {), I pushed it onto the stack. When I saw a closing bracket (),], }), I popped the top bracket from the stack and checked if it matched the current closing bracket. If it didn't match or if the stack was empty when a closing bracket appeared, the string was considered invalid. After checking all brackets, if the stack was empty, it meant the brackets were balanced and correct.

- **Descriptive Question (Q1):**
  Stacks are ideal for checking parentheses because they operate in "Last-In-First-Out" order. This matches perfectly with the rules for valid parentheses, as the last opened bracket must be closed first.

- **Time and Space Complexity (Q1):**
  The time complexity is O(n) because each bracket is examined only once. The space complexity is also O(n) because, in the worst case, the stack could store all brackets from the input.

Question 2: Postfix Expression

- **My Approach:**
  I also solved this problem using a stack. First, I created an empty stack and then split the postfix expression into numbers and operators. For each number, I pushed it onto the stack. When I encountered an operator (+, -, *, /), I popped the top two numbers from the stack, performed the required calculation, and pushed the result back onto the stack. After processing all elements of the expression, the last remaining number in the stack was the final result of the evaluated expression.

- **Descriptive Question:**
  Postfix notation is very useful because it does not require parentheses or complicated rules for operator precedence. Evaluating postfix expressions is straightforward: numbers are simply pushed onto the stack, and when an operator appears, calculations are done immediately using the top two numbers from the stack.

- **Time and Space Complexity:**
  The time complexity is O(n) because each token (number or operator) is processed exactly once. The space complexity is also O(n) because the stack may store most of the numbers in the expression before the operations are performed.

Manual Solution, Hand written:

Question 1: "[{()}]"

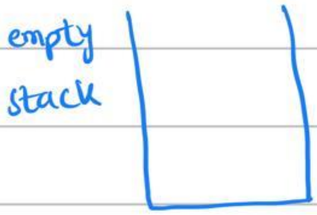
Step by Step Push/Pop

Step 1:

current char : — (start)

operation on stack: - creating empty stack

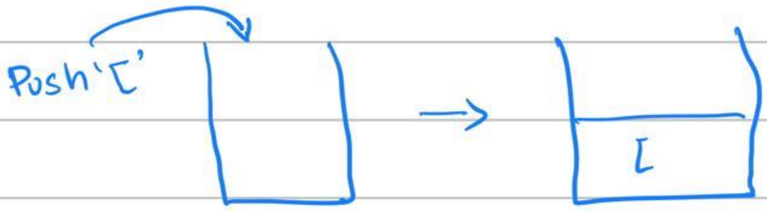
Stack state after step: []



Step 2: current char : [

operation on stack : push [

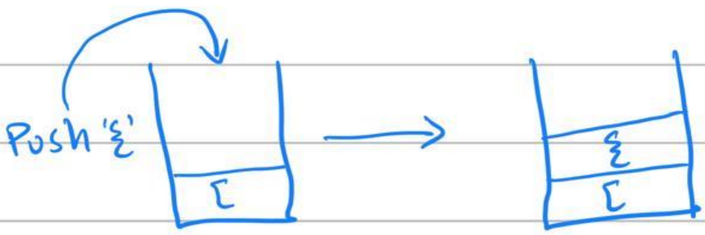
stack state after step: [



Step 3 : current char: {

operation on stack: push {

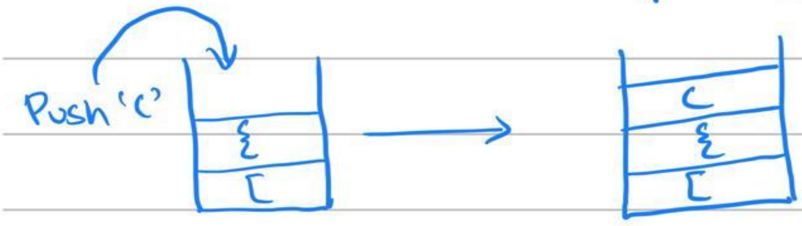
stack state after step : [, {



step 4: current char: (

operation stack: push `(`

stack after step: [, {, (

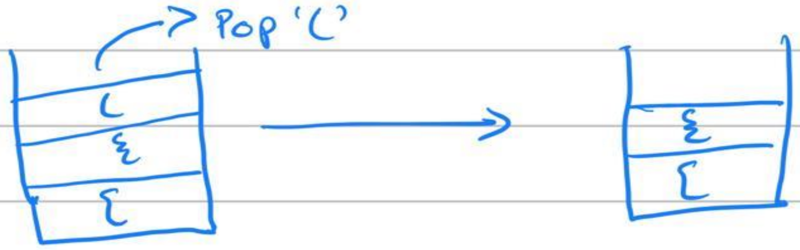


step 5: Pop ")"

current char:)

Operation stack: Pop '(' and compare

stack after step: [, {

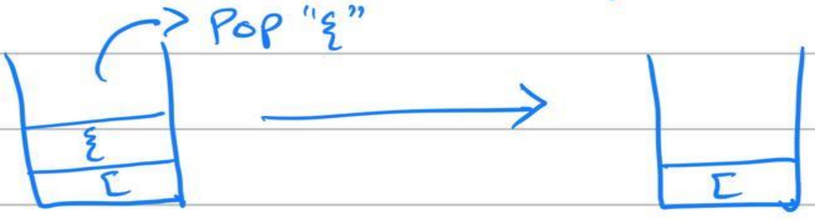


Step 6: Pop "}"

current char: }

Operation stack: Pop '{' and comparing

stack after step: [

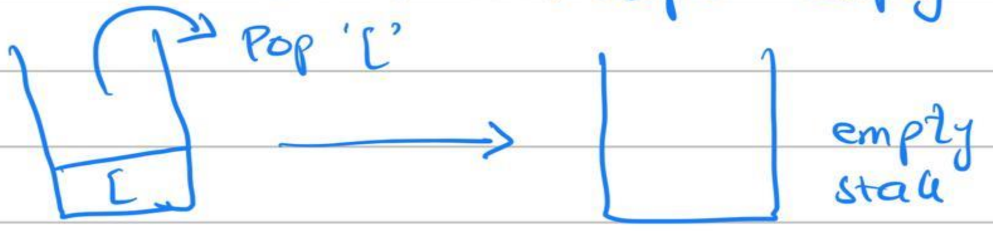


Step 7: Pop "]"

current char:]

Operation stack: Pop "]" and match

stack after step: empty


Pop '['
[
empty stack